ARP Request Support

Make it work the first time

Wed, May 21, 1997

When using Internet Protocols (IP), one must use the Address Resolution Protocol (ARP) to obtain a network physical address given a target IP address. Replies from ARP requests are cached in an ARP table so that ARP requests don't have to be performed too often. After some time, an ARP table entry is normally flushed, so that the next time it is necessary to send to that IP address, a new ARP request must be used. Although an IP implementation may hold back datagrams to be sent to a target IP address until it has obtained the hardware address, it is not required to do so. In many implementations, when faced with sending a datagram to a target IP address on the local network, the system sends an ARP request *instead* of the message, hoping that a high level retry will subsequently find a physical network address in the cache so that the datagram can be delivered. The IRM implementation of IP was designed to work in just this way. This note describes a method of implementing support for delaying a datagram until an ARP reply has been received so that ARP requests will occur automatically as needed without omitting the first datagram.

IRM Networking Background

IRM network support has had a long history. Network software organization in the preceding VME local station implementation even preceded IP support. It is designed around the communication of messages, as distinct from frames or datagrams. In both Classic and Acnet protocols, multiple messages that target the same node are concatenated into single frames or datagrams as can fit. (In some cases, messages cannot be combined, say in the case that a message is really the entire contents of a UDP datagram.) The concatenation of messages is handled by low level network code that is invoked after a network queue, one per network, has been loaded with pointers to network message blocks. Concatenation works with consecutively queued messages that target the same node and, in the case of UDP, the same node/port, or socket. Much of a front end's network communication consists of replies to data requests. A linked list of all outstanding data requests is maintained in an order such that multiple requests from the same node are grouped together. This scheme increases the likelihood that reply messages to multiple data requests from the same node that are due at the same time will be combined into a common frame or datagram for delivery. Often a number of IRMs are logically connected through a data server node, so that the chance of having multiple requests stemming from a common node is even more likely. Concatenation, of course, improves the efficiency of network utilization and is therefore deemed a "good thing."

The support for queuing all network messages is handled by a common routine called OUTPQX. It has the job of placing the pointer to a network message block into an OUTPQ ("output pointer queue") according to the target network. The target network is determined by the destination node# word found in the message block. Ranges of

Node# (hex)	*Network*
000x–00Ex	token ring via Acnet logical node table
00Fx	multicast raw
01xx–03xx	token ring raw
04xx	token ring raw
05xx–07xx	token ring raw *or* ethernet IP via DNS
08xx	token ring raw via token ring/ethernet bridge
09xx–10xx	token ring IP or ethernet IP
09Fx	multicast IP
7Axx	arcnet raw
8xxx	ethernet raw
6xxx	token ring UDP socket
Exxx	ethernet UDP socket

The range 05xx–07xx has a special significance. This range may imply use of token ring or ethernet, and it may be raw or IP. Node#s in this range will be sent via IP if the local node's global "broadcast" node# is in the range 09xx. (The "broadcast" node# is usually a multicast node# this is used to target requests that must be fulfilled by contributions from more than one other node, or to target Classic protocol device name lookup requests.) To get the IP address, the IP Node Address Table IPNAT is consulted that holds cached IP addresses that are derived from the local Domain Name Server via the local application DNSQ. All current IRMs are configured with a "broadcast" node of 09Fx. The choice of ethernet or token ring is made depending on CPU board and whether IP is to be used. On an MVME162 board, ethernet is always used for IP; ethernet is also used for non-IP (raw) when there is no token ring interface present. In practice, IRMs use ethernet on IP. MVME133 board-based systems use token ring, either IP or raw. Only a 162 board system can use both token ring and ethernet, and it will always use ethernet for IP.

IP communications is based upon a socket, which specifies an IP address and a UDP port#. (ICMP and IGMP effectively use a zero port#.) The low 12 bits of the socket node# in the above table refer to an 8-bit index into the ARP table and a 4-bit port# index. Each ARP table entry in active use has an associated port# block that can contain up to 15 active port#s from that node. In this way, a single word used as a target node# encodes a UDP socket. There are 254 ARP table entries available for sockets in active use, each of which can deal with up to 15 active UDP port#s.

New scheme for handling ARP requests

The networking support in IRMs has always been "in a hurry." An IRM is a front end that adheres to real-time performance. The notion of holding up network communications with other nodes while awaiting an ARP reply is not consistent with

realtime functionality. As a rule to be followed, it's "ok" to hold up messages that are to be sent to a target node for which an ARP request must be sent, but it's not "ok" to hold up messages to be sent to other nodes. The new scheme honors this rule.

New code was added to the OUTPQX routine to detect the case that an ARP request will frustrate delivery of a message using IP, if the message were allowed to pass on to the network software that builds frames. In such a case, an ARP request message is broadcast to the network, a ptr to the message block is queued in a data structure linked with the corresponding ARP table entry, and success is returned to the caller, implying that the message block has been "queued to the network." When the ARP reply is received, often within a few milliseconds, and the hardware address found therein is deposited into the corresponding ARP table entry, all the queued message block ptrs are passed through OUTPQX again, this time with assurance that they will really be queued to the network. If no ARP reply is forthcoming, after a couple of seconds, the message ptrs are passed through OUTPQX in such a way that they are queued to the appropriate OUTPQ, but they are marked so that the lower level network frame-building software will ignore them. As a result, they can still be handled by the Queue Monitor task, which has the responsibility of freeing blocks that are no longer needed following completion of transmission, or of marking them no longer busy so they are available for subsequent retry use.

An ARP queue block that is allocated to house the queued message block ptrs is large enough to hold 13 ptrs. If more are needed, another ARP queue block is allocated and linked to the first, so that there is no real limit to the number of messages that can be queued awaiting an ARP reply.

To implement this new ARP queue support, changes were made to OUTPQX as described above, to the IPARP suite of routines that support access to the ARP table, and to the SNAP Task, which handles ARP replies.

Classic Protocol

Message formats for the pedestrian

August 7, 1992

This note introduces the classic protocol message formats in use by the local control stations used at Fermilab in the D0 and Linac control systems. The messages can be used on token ring with SAP=$18, or they can be used from any IP node targetting UDP port# 6800 (decimal) for requests. For more extensive details, consult other documents.

Robert Goodwin
goodwin@fnal.gov

There are only a few basic classic protocol message type numbers as follows:

0. Data reply
1. (n.u.)
2. Data request
3. Data setting
4. Analog alarm
5. Digital alarm
6. Comment alarm

The following message formats are presented as arrays of 2-byte words in which the big endian byte order is assumed (hi byte first).
Hex data is used when possible in the format pictures. The message type number is found in the most significant 4 bits of the third word.

More than one message can be placed into a single UDP datagram.
The message size word in that case delineates each message.

The request message type is described before the reply.

Message type 2. Data request

```
+-----------------+
|  message size   |  message size (including this word) in bytes
+-----------------+
|  dest node#     |  node# to which this message sent.
+--------+--------+
|  2  s  | list#  |  $20= non-server. $28= server. list# identifies request.
+--------+--------+
| period | #ltyp  |  period in 15Hz cycles (0=one-shot), #listypes.
+----+---+--------+
| il |  #idents   |  ident length in bytes (4 bits), #idents (12 bits)
+----+---+--------+
| ltyp#  |  0  0  | \
+--------+--------+  |  repeat for #listypes
| #bytes req'd    |  |
+-----------------+ /
|  ident node#    | \
+-----------------+  |  repeat for #idents
|  ident index#   |  |
+-----------------+ /
```

The example shown is for the case of one listype, one ident, ident length=4. There are about 80 listypes defined. The meaning is roughly the kind of data that is desired. It implies a certain ident type and therefore ident length. Simple examples of listypes are:

listype#	kind of data requested	ident type
0	analog channel reading	channel#
1	analog channel setting	channel#
2	analog channel nominal value	channel#
3	analog channel tolerance value	channel#
4	analog channel alarm flags, count	channel#
20	memory data by bytes	address
21	digital bit I/O	bit#
25	digital byte I/O	byte#
29	memory data by words	address
40	analog reading in engineering units	channel#
41	analog setting in engineering units	channel#

The meaning of the request is that listype data is requested for all listypes using all idents given. For each listype and #bytes, generate response data for each ident. The ident array is processed in sequence for each listype. Thus a request is in general a matrix request. Therefore requests cannot be combined if one needs separate arrays of idents for each listype. In that case, more than one request should be made, each using a diffrent list#. If one wants 1 Hz readings and settings for channels 101,102,108 in node 576,

then the request would be as follows:

```
001E  size=30
0576  dest node
2033  non-server request using list#=$33
0F02  1 Hz, 2 listypes
4003  4-byte idents, 3 idents
0000  reading
0002  2 bytes
0100  setting
0002  2 bytes
0576  target node/chan list
0101
0576
0102
0576
0108
```

In this non-server case, the dest node of the request message must match the target node# in each ident. In the server case, the request will be sent (as a non-server message) by the dest node to the target node, or in the case of more than one node in the list of idents that is not the same as the dest node, to a group (multicast) destination address to reach all local stations. In the group case, only stations which match at least one of the nodes in the list of idents will respond at all, including only their own contribution. The separate contributions are arranged in original request order and returned in a single reply. For this to work best the stations should run synchronized.

Message type 0. Data Reply

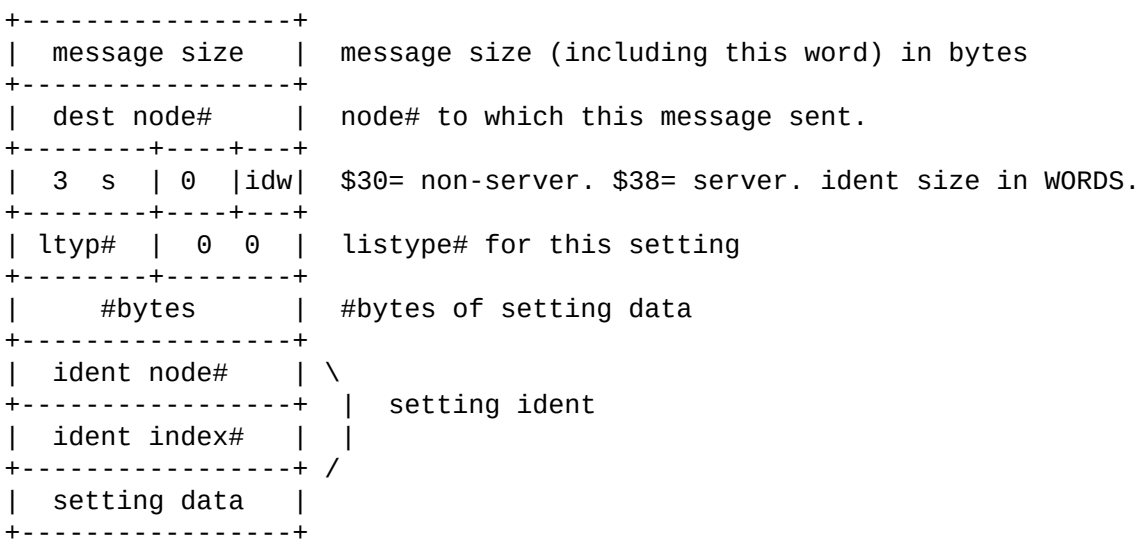
```
+-----------------+
|  message size   |  message size (including this word) in bytes
+-----------------+
|  dest node#     |  node# to which this message sent.
+--------+--------+
|  0  s  | list#  |  $00= non-server. $08= server. list# identifies request.
+--------+--------+
|   status word   |  reply status return code. 0=no error.
+-----------------+
|    reply data   |
+-----------------+
|        "        |
+-----------------+
```

This example shows the case of 4 data bytes returned. The dest node will be the node# of the requester node. For requesters that are not local stations there is no "node#", so this word will be zero. There is a single status word for the entire reply.

For the specific example given for the request message example, the following might be a reply:

```
0014  message size
0000  (zero dest node# for non-local station requester)
0033  reply type, non-server, list#=$33.
0000  status=0 (no errors)
1234  reading for chan 101
151D  reading for chan 102
4866  reading for chan 108
1230  setting for chan 101
1508  setting for chan 102
4882  setting for chan 108
```

Message type 3. Data setting



One setting message targets one device by listype# and ident. An example of a setting to channel $300 in station 576 would be as follows:

```
0010  size
0576  dest node
3002  setting, 2 word ident
0100  setting listype#
0002  2 bytes of data
0576  target node#
0300  channel 0300
1234  setting data value
```

In this non-server case, the dest node of the setting message must match the target node# in the ident. In the server case, the setting will be sent (as a non-server message) by the dest node to the target node.

Message type 4. Analog alarm

```
+-----------------+
|  message size   |  message size (including this word) in bytes
+-----------------+
|  dest node#     |  node# to which this message sent.
+--------+--------+
|  4  0  |  0  0  |  analog alarm
+--------+--------+
|    channel#     |  analog channel# in alarm
+-----------------+
|   alarm flags   |  flag bits. bit#8 (mask $0100) is good/bad bit. 1=bad.
+-----------------+
|    reading      |  analog reading
+-----------------+
|    setting      |  analog setting
+-----------------+
|    nominal      |  nominal value
+-----------------+
|    tolerance    |  tolerance value
+-----------------+
|    spare        |  spare word
+--------+--------+
|   'A'  |   'B'  |  analog 6-char channel name (example name='ABCDEF')
+--------+--------+
|   'C'  |   'D'  |
+-----------------+
|   'E'  |   'F'  |
+-----------------+
|  year  | month  |  time of alarm detected in BCD
+-----------------+
|  day   |  hour  |
+-----------------+
| minute | second |
+-----------------+
| cycle  |  0  0  |
+-----------------+
|                 |
+----fullscale----+  reading fullscale conversion constant (IEEE flt pt)
|                 |
+--------+--------+
|                 |
+-----offset------+  reading offset conversion constant  (IEEE flt pt)
|                 |
+--------+--------+
|   'V'  |   'O'  |  engineering units text (example='VOLT')
+-----------------+
|   'L'  |   'T'  |
+-----------------+
```

An analog alarm message is sent for any good/bad transition of an analog channel that is in the alarm scan (bit#15 of alarm flags word set). Examine bit#7 of the alarm flags word for a good(0) or bad(1) message. The analog alarm scan has a hysteresis to prevent chatter. Once a channel's reading has

drifted outside the tolerance window from the nominal value, and a "bad" alarm message is generated, the reading must be found within half a tolerance before a "good" message is generated.

Inclusion of the scaling constants permits calculation of the reading in engineering units at the time the alarm (good or bad) message was generated. The engineering units scaling is linear and uses the formula:

units = float(raw)*fullscale/32768. + offset

A specific example of an analog alarm message is:

```
002E  size
00F0  group dest node
4000  analog alarm
0150  channel 0150
B000  active(8000), inhibit(2000), two-times(1000), good(0100)
C596  reading
0000  setting (motor-controlled device has no setting value)
C5BC  nominal
0064  tolerance
0000  spare
4848  HHVOLT (H- preaccelerator high voltage)
564F
4C54
9208  time of alarm: 08/05/92 1648:03, cycle 14.
0516
4803
1400
44CC  fullscale  1638.
C000
0000  offset     0.
0000
4B56  units text 'KV  '
2020
```

The computed engineering units reading value in this case is -747.5 KV.

Message type 5. Digital alarm

```
+-----------------+
|  message size   |  message size (including this word) in bytes
+-----------------+
|  dest node#     |  node# to which this message sent.
+--------+--------+
|  5  0  |  0  0  |  digital alarm
+--------+--------+
|      bit#       |  digital bit# in alarm
+-----------------+
|   alarm flags   |  flag bits. bit#8 (mask $0100) is good/bad bit. 1=bad.
+-----------------+
|   'P'  |   'R'  |  digital 16-char bit text  (example='PREACC AIR COND ')
+--------+--------+
|   'E'  |   'A'  |
+-----------------+
|   'C'  |   'C'  |
+-----------------+
|   ' '  |   'A'  |
+--------+--------+
|   'I'  |   'R'  |
+-----------------+
|   ' '  |   'C'  |
+--------+--------+
|   'O'  |   'N'  |
+-----------------+
|   'D'  |   ' '  |
+-----------------+
|  year  | month  |  time of alarm detected in BCD
+-----------------+
|  day   |  hour  |
+-----------------+
| minute | second |
+-----------------+
| cycle  |  0  0  |
+-----------------+
```

A specific example of a digital alarm might be as follows:

```
      0022  size
      00F0  group dest node
      5000  digital alarm
      019C  bit 019C
      C100  active(8000), nominal=1(4000), bad(0100)
      4354  bit text (16 chars) 'CTF NTFREQ      '
      4620
      4E54
      4652
      4551
      2020
      2020
      2020
      9208  time of alarm: 08/07/92 1115:22, cycle 1.
      0711
      1522
      0100
```

Message type 6. Comment alarm

```
+-----------------+
|  message size   |  message size (including this word) in bytes
+-----------------+
|  dest node#     |  node# to which this message sent.
+--------+--------+
|  6  0  |  0  0  |  comment alarm
+--------+--------+
|    comment#     |  comment index# of alarm
+-----------------+
|   alarm flags   |  flag bits. (no good/bad bit for comments)
+-----------------+
|   'V'  |   'M'  |  comment 16-char text  (example='VME ALARM RESET ')
+--------+--------+
|   'E'  |   ' '  |
+-----------------+
|   'A'  |   'L'  |
+-----------------+
|   'A'  |   'R'  |
+--------+--------+
|   'M'  |   ' '  |
+-----------------+
|   'R'  |   'E'  |
+--------+--------+
|   'S'  |   'E'  |
+-----------------+
|   'T'  |   ' '  |
+-----------------+
|  year  | month  |  time of alarm detected in BCD
+-----------------+
|  day   |  hour  |
+-----------------+
| minute | second |
+-----------------+
| cycle  |  0  0  |
+-----------------+
```

There are only two comment alarms in current use. One is the system reset message, and the other is the alarms reset message. A specific example of a comment alarm might be as follows:

```
      0022  size
      00F0  group dest node
      6000  digital alarm
      0000  comment index 0
      8000  active(8000)
      564D  comment text (16 chars) 'VME SYSTEM RESET'
      4520
      5359
      5354
      454D
      2052
      4553
      4554
      9208  time of alarm: 08/07/92 1129:20, cycle 5.
      0711
      2920
```

IP Fragmentation

Implementation scheme

Mon, June 26, 1992

Internet Protocol (IP) specifies support for fragmentation and reassembly of packets to form datagrams. The implementation for IP support in the local stations must include this feature. This note discusses how it is done.

Maximum datagram size

Theoretically an IP datagram, including the IP header plus message, can be 64K bytes in length. In practice, however, some implementations put a smaller upper limit for reasons of memory constraints and economy. According to Douglas Comer, a general expert in the field, timesharing systems typically choose values in the range of 4K–8K bytes. (See "Internetwork ing with TCP/IP Volume II" page 32.) With the token ring 4Mb network, the hardware frame length limit is about 4K bytes, so it is natural to consider adopting a local limit of 4K bytes for the maximum datagram size. The ethernet frame size limit is 1500 bytes, which is too low to consider for a datagram size limitation.

A limit of about 2004 bytes is detectible sending "multinet ping" requests from the Vax. (It may only be a ping client program limit.) The Vax limit on datagram replies to ping requests is at least 5000 bytes, the limit of the local station's current software. The Vax's UDP echo server limit seems to be 4144 bytes.

MTU size

Separate from the issue of datagram size limit is the choice of Maxi mum Transmission Unit, or MTU. This is the limit on fragment size transmitted on a physical network. With ethernet networks this limit is almost always 1500 bytes, the same as the hardware frame limit. In the Sun that uses the SBus token ring board, the MTU seems to be 2044 bytes for the size of the IP datagram. From RFC-1122, it is recommended that one choose an MTU for use with foreign net works (destination network id different from local) that is 576 bytes, a generally accep table limit in use throughout the world-wide Internet. From tests with some far-away IP nodes, however, it seems that 1448 byte datagrams work without further fragmentation being imposed.

Whatever the local decision is regarding maximum datagram size and MTU, it is necessary to support reassembly of IP packet fragments. So this support is needed for the local station IP support software.

Reassembly

The local station uses the IPARP table to hold IP addresses and hardware addresses and related port#s. The IP address is the key to this table. An entry in this table is given by a pseudo node# word, which is an index to a table entry

and also identifies a port#; thus, it fulfills a function analogous to a socket.

When an IP fragment is received, specified by the MF (more fragments) bit set in the IP header and / or a nonzero fragment offset value, something special must be done with it; otherwise, it simply remains in the DMA buffer written into by the chipset to await higher level processing. To support fragments, it is necessary to use a timeout in case not all fragments arrive that are to build a datagram. After each fragment is received, the timer should be reset. A timeout commonly in use is about 60 seconds. After a timeout, the fragments must be discarded. (If frag ment #0 has been received at that time, then an ICMP "time exceeded" error mes sage is sent to the source host. See RFC-1122.) The point is that fragments may have to wait a long time before they can be built into a datagram, so they can not remain in the 64K DMA buffer, which can wrap in a second or so under heavy traffic. Allocated memory can hold the fragment for later reassembly.

It is not convenient to build the datagram as the fragments are received, because the total size of the expected datagram is unknown until the last fragment has been received that has a zero MF bit. If one allocates a datagram buffer of maximum size, however, it should be possible to do assemble the packets as they are received. To make it more interesting, the order of fragment arrival is not guaranteed, although this is not expected to be a problem within Fermilab.

If the maximum size buffer allocation is not used, then one can build a linked list for each datagram whose fragments are in the process of being collected. Frag ments belong to the same datagram if they have the same source IP address and the same identification word in the fragment's IP header. The linked list can be maintained in order of fragment offset to simplify detection of the point at which all frag ments have been received. When the datagram is complete, another area of memory can be used to hold the complete datagram, as higher level process ing must see the entire datagram in contiguous memory.

In the local station implementation, the SNAP Task handles IP received packet processing. If the routine that processes a fragment detects a completed data gram, it assembles it and sends a reference message about it to the SNAP message queue, just as the token receive interrupt routine normally does when it receives a SNAP frame. (The SNAP protocol SAP# $AA is used currently only for IP and ARP packets on token ring.) Then SNAP will notice a larger-than-normal datagram that is unfragmented and process it normally. For either ICMP or UDP datagrams, the checksum will be done on the entire datagram.

Fragmentation

Although gateways will fragment datagrams as needed, it is considered kind for hosts to fragment outgoing datagrams so that gateways can pass the fragments without having to further trim their sizes. Fragmentation support is

required to support datagram sizes that exceed the local network frame size limit.

Conceptually the matter of fragmentation is easier than reassembly. When a datagram is prepared to be sent, the transmit logic decides, based upon the destination IP address, that an MTU should be applied that is smaller than the datagram size. In this case, separate frames are built for each fragment, each one with an IP header. Each is queued in turn to the network hardware.

When fragmentation is applied, the message part of the fragment—except for the final one—must be a multiple of 8 bytes in size. (This is required because the fragment offset field in the IP header is expressed in 8-byte units.) Since the IP header part of the fragment is typically 20 bytes long, the maximum fragment size may be slightly smaller than the MTU. (This insures that the implementer is paying attention.)

The NetXmit routine assembles queued network messages into frames for delivery to the network hardware. When it has prepared the frame completely, it can make the decision on fragmentation. If the network id in the destination IP address is the local network, a different MTU is used from that used when the network is foreign. Use of different MTU values for both the local subnet and the rest of the Fermilab network is also supported.

Since the complete datagram is already assembled in the frame buffer ready to be transmitted before it is determined that it must be fragmented, the first fragment can be sent from the first part of the datagram, after adjusting for the fragment size in the IP header, setting the MF bit, and recalculating the IP checksum. The additional fragments can be built into new frame buffer space by copying the frame/IP/SNAP headers and the next fragment-size part of the message.

IPARP Table

IP global data structure

Wed, June 29, 1992

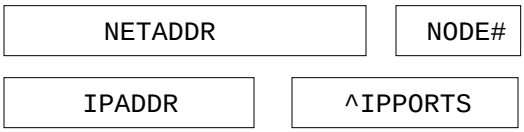
Every network node that supports Internet Protocol (IP) communications maintains an ARP table to relate IP addresses to hardware addresses. In the case of the local station's IP support, it also provides the "node#" that is used to reference that node (if it is a front end data source), and the port#s in use by that node. A "pseudo node#" is used to refer to a port of a node. It provides a functionality similar to that of an internet socket. This note describes how the IPARP table is used by the local station system to support IP.

```
Procedure InzIPARP;
```

At reset time, if no 'IP' exists in the first word of the IPARP table, which is system table#28, initialize the table and clear all entries; otherwise, clear the port# block ptrs in all entries. Although IP addresses are saved across resets, port# assignments for pseudo node#s are not. If the table header includes a nonzero IP address and netMask, then establish a ptr to the IPARP table in a system global variable. Note that when the table is first established and all entries are cleared, the header is also cleared. The local station's IP address and network mask must be entered manually and the system reset again, in order to enable IP support. See installation section.

```
Function PsNIPARP(port: Integer; ipA: Longint; VAR netA: NetAddrType):
Integer;
```

This function is used when an IP (or ARP) datagram is received and processed by the SNAP task. If it is a UDP datagram, the source port# is specified in the call; otherwise, port#0 is used. The table is searched for a match on the source IP address (or sender's IP address in the case of an ARP message), and the hardware address is updated there. If it is not in the table, a new entry is added to the table to hold this information. If there is no allocated port# block for the entry, then one is allocated. For UDP, the port# is installed in the port# block, if it is not already there. The returned value is the pseudo node# that is used to reference the entry. The format of the pseudo node# is chosen so that it does not conflict with node#s in current use among accelerator nodes. This uniqueness is used to denote that IP encapsulation is needed when sending the message on the network, and by reference to the IPARP table entry, the parameters of that encapsulation.The current form of the pseudo node# is $6nnp, where nn is the table entry# in the range 2–255, and p is the port# index in the range 0–15. (For nonzero port#s the index range is 1–15.) In the case of an error, a zero is returned. The IPARP table entry is 16 bytes as follows:




```
Function GetIPARP(pNode: Integer; VAR port: Integer; VAR node: Integer;
VAR ipA: Longint; VAR netA: NetAddrType): Integer;
```

When a frame is due to be transmitted, and the frame header is being built, this function retrieves the IP address, network address, node# and port#, from the IPARP table entry specified by the pseudo node#. These values are used to build the IP header and UDP header of the datagram. The node# is used to replace the pseudo node# used as the destination node field in the acnet header and classic protocol cases.

```
Function IncIPARP(pNode: Integer): Integer;
Function DecIPARP(pNode: Integer): Integer;
```

For each port# registered in the port# block associated with a given IP node, there is a use count. These two functions operate on the use count pointed to by the given pseudo node#, which provides for support of multiple requests from a given UDP source port. When a data request is initialized, and a ptr to its request memory block is inserted into the chain of active requests, IncIPARP is used to advance the use count associated with the target port for the reply. When the request is cancelled, and the request is removed from the active chain, then DecIPARP is used to reduce the use count. In this way, the available 15 port# slots associated with an IP node can be reused as needed.

```
Function NodIPARP(pNode: Integer; node: Integer): Integer;
```

The concept of a node# word is used in both acnet and classic data request protocols. At the time PsNIPARP is called by the SNAP task, any node# specified in the message is unknown. When higher level acnet protocol handling determines what the source node# is, then it can use this function to update the node# word in the IPARP table entry.

```
Procedure TimIPARP;
```

Every second, the QMonitor Task calls this routine to perform timeout logic on IP communications. TimIPARP scans all active IPARP entries and counts down the timeout word in the port# block header. The timeout word is reset

to a large value (currently 4000 seconds) every time the port# block is referenced by PsNIPARP or GetIPARP calls. Thus, when it reaches zero, it means no ports have been used for a long time with that IP node, so the block is released.

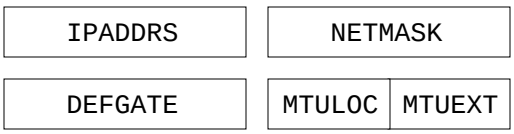
With the implementation of fragmentation and reassembly, additional timeout logic has been included in TimIPARP. When IP fragments are received, they are timed out in case not all fragments are received to make a complete datagram. Each time a fragment is received that is part of a given datagram, a timeout count is reset to a large value, which is currently 60 seconds. After that time, the fragment blocks are released, and a “time exceeded” ICMP error message is returned to the sending host.

When all IP fragments have been received for a given datagram, a complete datagram is built and passed to the SNAP Task via its message queue, just as if the entire datagram were received from the network. The block containing the complete datagram is timed out in case its associated message counter is not reduced to zero. (The message count word should be decremented by every program or task when it is finished processing the message; this can be done automatically by NetRead, NetRecv, UDPRead, UDPRecv, or by Classic Task processing, depending on the protocol used.) The current value for timing out completed datagram blocks that are unused is 60 seconds. Normally, the datagram block is released within one second after the message count word is reduced to zero by the next call to TimIPARP.

```
Function FrgIPARP(pNode: Integer; dIdent: Integer;
   fragPtr: FBlkPtr; VAR dgPtr: DgPtrType); Integer;
```

This routine is called by the SNAP Task to handle IP fragment processing. The pNode parameter is the pseudo node# returned from a call to PsNIPARP. The dIdent is the identification field from the fragment IP header that identifies the datagram of which it is only a fragment. The fragPtr argument is a ptr to a fragment block that holds a copy of the received fragment. The dgPtr variable is set to a ptr to the completed datagram block in the case that this fragment makes the datagram complete. See the document Fragmentation and Reassembly for more details on this.

The IPARP table header format is as follows:



The local station IP address must be entered into the table manually. Since IPARP is in non-volatile memory, it should not thereafter need to be changed. In addition, the subnet mask is kept there and an IP address of the optional default gateway. When a datagram is to be sent to an IP address destination for which there is no known hardware address,, a check is made using the local station IP address and the subnet mask to determine whether the target IP node is on the same subnetwork. If it is, then an ARP request message is sent to obtain the hardware address. If it is not, then the default gateway IP address is used to look up the gateway's hardware address. (If there is none as yet, an ARP request is sent to obtain it.) The message is then sent to the gateway's hardware address.

At the time this is written, when an ARP request must be sent, the datagram to be sent to the target node is discarded. Since the IPARP table entries are not timed out, this is not expected to be a problem. If it is, then a means of queuing up, and timing out, datagrams awaiting ARP responses must be found. With no means of timing out IPARP entries, it is possible for stale information to accumulate there. If a node changes its hardware address, for example, and it is on the same subnet as the local station, then the table entry would have to be manually cleared for that IP node. Again, if this causes problems, we may have to implement a time out for the IPARP table entries. Both this timeout and the queuing of datagrams awaiting ARP responses should probably be implemented at the same time.

It is hoped that 254 table entries, which means 254 IP addresses, will be enough for the local stations to keep track of. If it is not, then this would be another reason to implement a timeout on the IPARP table entries.

With no timeout of IPARP table entries, it is still ok for a station to change its hardware address, if that station sends a request to the local station. The hardware address used in the request message will update the IPARP table entry for that same IP address. If the change is in the IP address, a new entry will be automatically added to the table. If an IPARP table entry is cleared manually, the entry will be available for re-use.

Installation

When installing IP support for the first time, install the table #28 as 256 entries of 16 bytes each in the system table directory. After reset, the table will be initialized, but no IP address will be in its header. Install the IP address, subnet mask, and the default gateway address. Reset again. One other important item: install the SAP table entry#2 as $AA, or no SNAP (IP) frames can be received!

IP Multicast Addresses

Catalog of those in use for IRMs

Wed, Jun 18, 1997

A range of IP multicast addresses was assigned by networking for use by IRMs here at Fermilab. They are 239.128.02.xxx. The software in IRMs supports up to 15 of these, plus broadcast. They are selected for transmission by node#s 09Fx, where 09FF always means broadcast as used for ARP requests. This table shows the current usage of such addresses:

```
09F0  PET alarms
09F1
09F2  A0 alarms
09F3
09F4
09F5
09F6
09F7
09F8
09F9  All IRM nodes
09FA  All PET nodes
09FB  All BRF nodes (Booster HLRF)
09FC  All A0 nodes (Tesla/Photo-Injector)
09FD
09FE
09FF  (broadcast)
```

In the non-volatile memory of IRMs there is a table of 16 entries that corresponds to the above list. Each 8-byte entry contains a multicast hardware address and a diagnostic count of the number of frames transmitted to that destination multicast address. As an example, here is the table from a PET node:

```
0584:405B80 0100 5E00 02F0 0000
0584:405B88 0000 0000 0000 0000
0584:405B90 0000 0000 0000 0000
0584:405B98 0000 0000 0000 0000
0584:405BA0 0000 0000 0000 0000
0584:405BA8 0000 0000 0000 0000
0584:405BB0 0000 0000 0000 0000
0584:405BB8 0000 0000 0000 0000
0584:405BC0 0000 0000 0000 0000
0584:405BC8 0100 5E00 02F9 0000
0584:405BD0 0100 5E00 02FA 0000
0584:405BD8 0000 0000 0000 0000
0584:405BE0 0000 0000 0000 0000
0584:405BE8 0000 0000 0000 0000
0584:405BF0 0000 0000 0000 0000
0584:405BF8 FFFF FFFF FFFF 038B
```

The ethernet multicast addresses used conform to the ethernet convention for IP multicast. The upper 25 bits are fixed to the value 0100 5E as shown. The lower 23 bits match the lower 23 bits of the corresponding IP multicast address. In our case, this is 00-02Fx. As a specific example, this node can access all other PET nodes by

The IP multicast (Class D) address used would be `239.128.2.250`, or `$EF8002FA`. In order for a node to be able to use a given IP multicast address for *transmission,* the corresponding ethernet multicast address must be installed in the above table.

For a node to be able to *receive* from a given multicast address, the address must be in a different table that has room for up to 8 different ethernet multicast addresses. Looking at the same node used in the above example, we have:

```
0584:405240 0100 5E00 0001 00BB
0584:405248 0100 5E00 02F9 00D4
0584:405250 0100 5E00 02FA 00C7
0584:405258 0000 0000 0000 0000
0584:405260 0000 0000 0000 0000
0584:405268 0000 0000 0000 0000
0584:405270 0000 0000 0000 0000
0584:405278 0000 0000 0000 0000
```

In order to participate in the IGMP protocol that is used by multicast routers to determine whether to pass multicast IP datagrams onto its connected networks, each node must listen for the "all hosts" multicast address `224.0.0.1`, which uses the ethernet address `0100 5E00 0001` in the first entry of this table. The last two bytes in each 8-byte entry are a delay counter byte and a diagnostic counter that counts the reception of frames addressed to the given ethernet multicast address. Once each minute, the multicast router sends an IGMP request message to the "all hosts" address that asks the question, "what IP multicast addresses are of interest for the nodes on this network?" Each node that receives this request schedules an IGMP reply message to be sent after a random delay over the following 10 seconds. This is done for each multicast address in this table (but not for the "all hosts" address). The delay counter byte is set to a random value for each entry in use. For IRMs, the delay function is provided by decrementing this byte each 15 Hz cycle. If the counter byte reaches zero, an IGMP reply is sent to the same multicast address. (The router will see this because it can see all multicast frames.) Because the reply is targeted to the same multicast address in which the node is announcing its interest, any other node on the connected network that has the same interest will also receive the reply message; as it does so, it will cancel its own intent to transmit the same reply. In this way, minimal network traffic is required to keep the router up-to-date on which multicast addresses are needed on that connected network. (The router only needs to know if *any* node on a connected network has an interest in a given multicast address. It doesn't care which ones or even how many such nodes there are. One node's interest is enough to force the router to pass such addressed IP datagrams.)

Entries to be added to the above non-volatile table of multicast addresses enabled for reception are made manually as a part of system configuration. Changes to the

table may be made during system operation, and such changes will become effective right away. The system maintains a checksum of the multicast addresses in the table that it calculates about once a second. If it notices a change in the computed checksum value, it builds a new Multicast Setup command and sends it to the ethernet controller chip, an Intel 82596 on the MVME162 board. When it does so, it also sends a diagnostic Dump command that causes the chip to write a 300-byte record of various internal information it keeps. Within this block can be found the contents of the 64-bit hash register that is used for filtering multicast-addressed frames that the controller chip sees on the network. As soon as the chip detects a destination multicast address at the start of an ethernet frame, it hashes the 48-bit address into a 6-bit value. If the corresponding bit in the hash register is set, the frame is accepted, else it is rejected. Because this scheme permits reception of multicast addresses that may only coincidentally hash to the same 6-bit code of another that is in use, the network software must check that a received multicast frame is really addressed to a multicast address of interest, ignoring it if it is not. To that end, the above table is scanned by the ethernet receive interrupt software for multicast-addressed frames. But in order to derive maximum benefit of chip-level filtering of multicast frames, one may prefer to select multicast addresses to be used in a given installation so that they do *not* correspond to matching 6-bit hash codes. To do this, it is useful to examine the hash register contents in the 300-byte Dump area at `$1600C0`. Here is the beginning of that area in the same node above:

```
0584:1600C0 7F2E 0060 00F2 0000
0584:1600C8 40FF 003F FFFF 0240
0584:1600D0 0000 0584 0020 83B7
0584:1600D8 3117 9A3F C228 0000
0584:1600E0 0400 42A6 1220 1000
0584:1600E8 0000 0000 1100 D586
0584:1600F0 05FC 0C00 FFFF FFFF
0584:1600F8 0000 0000 00C0 0016
```

The hash table register is located at offset $26, where we find `1000 0000 0000 1100`. Thus, the register has three bits set, so that each multicast address uses a different hash code.

Specification of common usage of multicast addresses in IRMs is done via two node#s in yet another non-volatile table. The first is the "broadcast" node# that is used when an IRM must perform a device name lookup when it cannot find the name in its own device tables. This same node# is also used when an IRM must send a data request that is to be targeted to more than one other node. (Using multicast for this means that only a single message need be sent, rather than one to each node represented in the request message.) The second node# that may be specified is the target node# for Classic protocol alarm messages. Again, as an example, here is what is in that table for the above PET node:

```
0584:402000 0A00 D400 09FA 09F0
```

The "broadcast" node#, at `$402004`, is `09FA`.
The alarms target node#, at `$402006`, is `09F0`.

By configuring sets of nodes to use different multicast addresses, we can permit each set of nodes to operate in its own network world. This means that IRMs from different sets may use the same 60-character device names, for example. It also means that when the "broadcast" node# is used to send requests for devices from multiple other nodes, such requests will be seen only by the nodes within a set. For example, PET device names could match Booster device names, if desired. A node in the A0 set would not be constrained to use device names that are neither used in Booster HLRF nor in PET. And PET alarm messages received by an alarm handler will not have to filter out Booster HLRF alarm messages. IRMs have found to be generally useful front ends that can serve in various projects, even independent of Acnet. Some of these front ends are ultimately shipped to different labs even in other countries.

IP Security

"Safe sets" for the local stations

Tue, Jun 30, 1992

Introduction

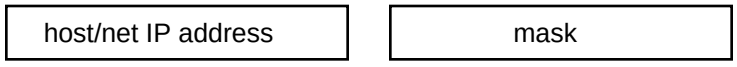
With the addition of IP support for the local stations, the issue of protection from unexpected settings takes on a new dimension. The Internet is a world-wide network; thus it is possible to access the local stations from anywhere on earth. While reading access presents no problem at this time, setting access is not desired. This note discusses a means of restricting setting access to the local stations while allowing unrestricted reading access.

Router blocking

One means of protection would deny IP datagram routing to any local station from outside Fermilab. This would certainly provide protection, but it also denies reading access, since Internet Protocol has no concept of a setting message as we use it in accelerator control. Only higher level layers of the control system proto cols recognize the distinction between a request for data and a setting to a device. Use of a router-based scheme may also be inflexible and difficult to maintain.

Trusted host solution

In unix systems, there is a file called "`/etc/hosts.equiv`" that contains a list of nodes that are "trusted" to perform operations that may be denied to other hosts. In the same spirit, the local stations may keep a table of trusted hosts that would be allowed setting access to devices. As the total number of hosts may be large, it may be more convenient to allow access by subnetworks to reduce the size of the table and the processing required to interpret it. A table entry may look like this:



When a setting is to be performed in response to an IP-based network message in any of the three device protocols (Classic, D0, Accelerator), the entries in this table are scanned. For each nonzero entry, the source IP address (from the IP header of the setting message) is EOR'd with the host/IP address field and AND'd with the mask field. In this way, entire networks or subnetworks can be repre sen ted by a single table entry. Setting access by individual hosts can also be accom modated using a mask of all ones.

Diagnostics

The implementation of setting limitations may provide a chance to include diagnostic information as well. Counts of the number of settings accepted via IP-based access can be kept for the three device protocols supported: Classic, D0

and Accelerator. In addition, the IP source address used by the last such setting and the time-of-day could also be included. Since this is a security issue, one could also keep the last IP address from which a setting was denied, along with the time-of-day that it was attempted. A 64-byte entry for this info:

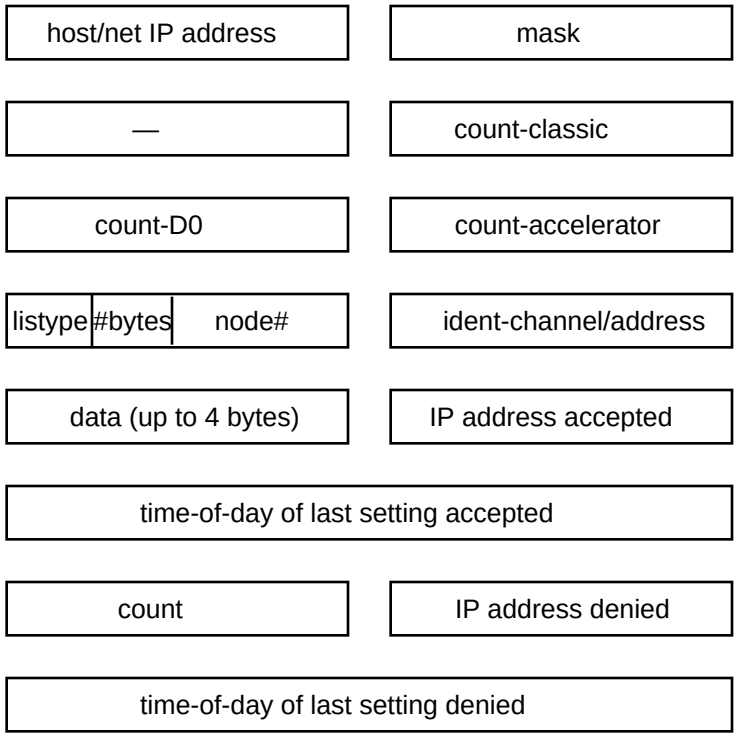


Table maintenance

Using the above data structure, it is obvious that only the first two fields are static; the rest are dynamic. It is likely that many local stations will have the same table (static part). This stems from the fact that network access to one local station has always given access to any other. As we download local applications, we can also download the static part of this table. To maintain it and be able to edit the table requires access to the appropriate software development tools such as MPW. Making a change would require the same effort as changing a local appli cation program. The HELPLOOP "text file," providing the prompt text for config uring parameters of local applications, was implemented in this way, although it was installed in only one station, accessible from the rest. Use of group addres sing makes it easy to update the table in, for example, all Linac stations at once.

IP Support

Internet protocols for local stations

8/28/97

Introduction

In recent years, the internet protocol standard has grown by leaps and bounds. Because support for it is included with every workstation, it is natural to consider providing this support for the local station systems, in order to make it easier to write support for data requests and settings to a control system. This note is a working document that assumes some TCP/IP familiarity.

Note that internet protocol support is only a beginning for a host; the data request protocols must be built on top of any internet protocol used. The fundamental node-node communications is supported by IP, Internet Protocol. In order to make the network accessible to a user, an entity *within* a node, a higher level is needed. Above IP, the TCP/IP protocol suite includes two basic types of data communications: stream-oriented (TCP) and datagram-oriented (UDP). Each of these types is suited for different applications, yet higher level protocols.

The stream-oriented protocol (TCP) refers to a byte stream of data. Network messages that support the byte stream have no built-in boundaries at all, so a user of TCP must build that into the higher layers. It does, however, insure that the byte stream arrives in the same order as the byte stream that was sent, by use of sliding window acknowledgements. If it were important to support Telnet, FTP, or SMTP with the local control stations, then TCP support would be required.

The datagram-oriented protocol (UDP) is used to transport a record, called a datagram, across a network. It is sent on a best efforts basis, with no guarantee that the datagram arrives at the receiving node nor that a succession of data grams arrives in order. For a control system, which is fundamentally record-oriented, the datagram approach has appeal. The lack of a guarantee that the datagram reached its destination node is mitigated by the reply to a data request, or the acknow ledg ment to a setting. The lack of guarantee of the order is really only a problem for very large internets and is unlikely to be a problem in practice within a single laboratory. (If one were attempting to do 15 Hz control from Sweden, for example, there might be need for more concern.) Protocols that expect to use UDP are NFS, RPC, SNMP, and TFTP.

In view of the above arguments, assume that a data request/setting protocol is built on top of UDP, the datagram protocol.

Token ring frame format for IP

What is the frame format of IP datagrams on the token ring network? According to RFC-1042, the format uses the SNAP variation of the 802.2 header. This means that the DSAP and SSAP are $AA, the control byte is $03 (UI), the next three bytes are the organization code $000000, and the last two bytes are the same as the Ethernet type word. For IP frames, this is $0800. For ARP, it is $0806.

IP datagrams

What does the IP layer do? Its job is to send a datagram, limited to 64K bytes in length, to another node across an internet. This might imply that fragmen tation would occur. So the IP Task must support fragmentation and reassembly. In the current local station software, there is convenient support for frame *reception* passed to a receiving task. If no fragmentation takes place, then the current support for network messages largely takes care of itself, assuming that the protocols within the UDP header are the same as those handled now. If fragmentation occurs, so that a fragment is received that does not represent the entire datagram, then the fragment must be copied into a datagram buffer of larger size. This will make it slower, of course, but is probably unavoidable. When the complete datagram has been assembled, it can be passed to the destination task. As a first step in support, one could ignore fragmen tation on the token ring network. Data requests/replies have, until now, always been limited to about 4K bytes, the maximum frame size for token ring. (Since this was first written, fragmentation support was added 6/23/92.)

For frame *transmission*, there must be some means of denoting that IP packaging is required. This must include a way to keep the port# of the requester as well as the requesting node. A pseudo-node# can be used as the source node of the request that can help recover the original requester's UDP source port#.

The idea here is that support for the present suite of protocols remains and is only enhanced by the addition of IP and UDP support. These provides an optional wrapper for the usual protocols. To support both Classic and Acnet-header-based protocols, two different well-known UDP server ports are used.

Address resolution

What about ARP? The Address Resolution Protocol is used to find the hardware address that corresponds to a given IP address in the case that the IP address is on the same subnetwork. This request is broadcast with a special value for the type word ($0806), in hopes that a node will answer with the hardware address that corresponds to the given IP address.

What about ARP processing? If we need to send an ARP, it's one more comp lication, because it means that the frame cannot be transmitted until a reply is returned from the ARP request. An easier way is to use a table that must be filled

by a node sending a message at some time to the given node. After that, it is registered, and its hardware/IP address pair is known. A front end should not often worry about sending a message to a hitherto unknown node.

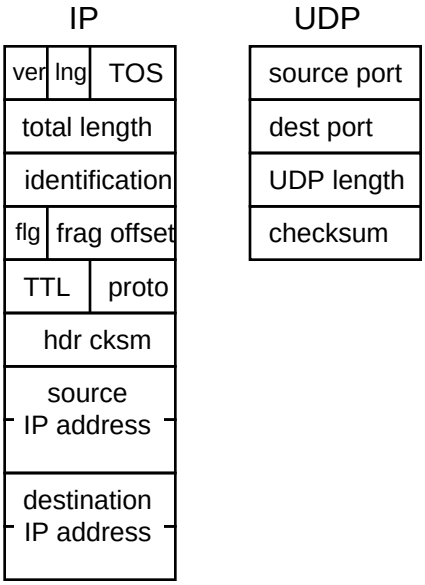
The ARP table entries are usually timed out, in order to accommodate changes in network addresses. If this is done, it will become important to support queuing of datagrams awaiting ARP replies. At first, we can keep ARP table entries forever and not support datagram queuing.

IP addresses

When a message is received, a node# must somehow be assigned to it. If the requesting node does not use the `40020000ttnn` convention, then it might be an Ethernet console with node# `08xx` using the `55002000yy17` convention, in which the yy is the bit-reversed value of `nn`. There is a table of Ethernet addresses that correspond to `0800-08EF`. If we can use some of that space for dynamically assigned IP requesters, it would provide a natural place to store the hardware address. But what about the IP address? When NetXmit has a `08zz` node#, it can find the right hardware address, say, but where can it find the IP address? Of course, it was in the request message, but where can it be stored for later retrieval? A different table should be used for this, one that is kept in non-volatile memory and filled whenever a frame is received from an IP address.

When a frame is received that uses IP or ARP protocols, capture the hardware address from the frame header and the IP address from the IP header and the UDP port# from the UDP header. Search the table for a match on these values. If none is found, install a new entry. Assign an *internal* node# and replace the source node# in the acnet header with it. In this way, a request message can be processed normally and the reply queued to the network using this internal node# as a destination. The NetXmit logic can use it as a signal to send an IP datagram and to recover the other information for building the frame header. See the document IPARP Table for more discussion on the table's implementation.

The IP and UDP headers have the following formats:

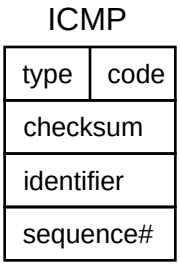


The *ver* is the IP protocol header version number $04, the *lng* is the number of longwords in the header, between 5 and 15, but usually 5. The *type-of-service* byte can be ignored initially and built as a constant (0) for transmission. The *total length* is the #bytes in the datagram, including both the IP header and the rest of the datagram. The *identification* is a word that is a sequence# of IP datagram sent by the source node. It has the same value in all fragments of a datagram. The *flg* bits and *frag offset* are used for fragmentation. The *don't fragment* flag bit prevents a gateway from forwarding a fragmented datagram. A *more fragments* flag bit indicates that this fragment is not the last one. It is only when a fragment of a datagram that contains the *more fragments* bit clear that IP learns the length of the entire datagram.

ICMP support

This is the error reporting mechanism. It is based upon IP just as UDP is, but it is a required part of IP support. It provides a “ping” echo service, for one. Error messages should be directed to the original source node in general, since the route taken by the message in error is not available. Care should be taken to use ICMP error replies only in situations specified by RFC-1122 recommendations.

The format of the ICMP message is as follows:



Network messages in local station

Local station network handling is a higher level of support than either IP or UDP, in the sense that it is message-based and not frame-based. An application using the network routines does not see the frame boundaries, although it can flush the network queue to force a frame boundary. It usually deals with messages that it receives from (or queues to) the network. All current network-related applications have this message-oriented view. Also, they can generate either Classic protocol messages or Acnet protocol messages. It is desirable to *not* break this mechanism by the intro duction of IP support. This means that frame transmission using IP datagram frames must be automatically determined by NetXmit logic, which extracts all queued network messages, combines them into frames and hands them over to the token ring chipset hardware.

Frame formats used by local station

How can NetXmit decide what type of frame header to use? The difference between Classic and Acnet is decided by the memory block type# that holds the queued message. But we also need to communicate using ARP and ICMP, so these may require an additional memory block type# that can be queued to the network that would cause NetXmit to formulate those frames appropriately. Also, one must insure that ARP and ICMP messages are not combined with others in a common frame, as no host will expect it. For the UDP frames we are discussing, multiple messages *can* be supported in the same frame because, to a host that receives such a frame, it is only a single message sent to a server UDP port. The port handling logic will identify the messages of, say, Acnet format and dispatch them to the appropriate network tasks.

Fast Time Plot Data Acquisition

Faster than 15 Hz

Fri, Jan 14, 1994

The analog IndustryPack module used in the Internet Rack Monitor (IRM) includes support for recording 1 KHz samples of 64 A/D channels in a 64K byte circular buffer. The buffer wraps in 512 ms. It is desired to access such data for the purpose of making plots. In the Acnet control system, the Fast Time Plot protocol (FTPMAN) is used to acquire this data from the front ends. A console requests data to be delivered at 15 Hz down to 2 Hz. In principle, the console could collect 1 KHz data, but it limits its Continuous mode plotting support to 720 Hz data. In the "Auto" mode, the limit is 200 Hz. Faster rates must be accessed via the Snapshot mode.

The SSDN that is sent in FTP's Timing Info Request message is the same SSDN that is used when the named device is called up for display on a Parameter Page. In order to access the data from the analog IP board, FTPMAN must somehow derive the class code that describes what plotting support is allowed for the given signal. To deliver the plotting data, both the 64K memory that holds the 1 KHz data as well as the register block for the IP board must also be determined. But the SSDN now contains only the analog channel number that points to the data pool.

One possibility would be to utilize a spare byte that has so far not been used in the SSDN data structure. This would require changing the off-line uploading program that updates the Acnet database with changes made in a local station. How can the uploading program determine in what set of 64 channels a given channel resides, if indeed it resides in *any* set of 64 channels from an analog IP board. One could use a spare byte in the analog descriptor for this purpose. And one could assume that the low 6 bits of the channel number indicate which channel out of a 64-channel set is being referenced. It would seem wasteful to use the spare byte for this purpose, however, as each channel in a 64-channel range would have to be marked in the same way. Perhaps a small table of channel # ranges could relate to the memory and register block locations.

A simpler approach might be to fix the ranges of channels used for available IP boards. An IRM can have one or two such boards. But a VME-based station that uses IP carrier boards might have more.

Another concern is to provide support for up to 1 KHz data via a new listype. In this case, an ident format must be defined. It could include the type of 64-channel block. But how can the user enter this information, say, on the Macintosh Parameter Page? The support code for this listype could return data in a format similar to that used for data stream returned data. The first word of the reply data could be the number of data values returned for each ident. The internal pointer could include the 16-bit offset of the next datum to be collected, so that each update would begin

where the last one ended in access to the circular memory buffer.

A simple approach can be as follows: Assume that channel#s `0100–013F` are supported by the IP module in slot *d*, in which the base address of the memory is `00630000` and the register block is at `FFF58300`. For channel#s in the range `0140–017F`, assume the IP module is in slot *c*, the memory base address is `00620000` and the register block location is `FFF58200`. For VME-based stations that do not use IP modules, these channels could not be assigned; at least no one could try to FTP them. This scheme could work for IRMs that have 64 or 128 channels.

In order to collect data that is measured at times relative to clock events, a different plan should be adopted. If the reply data includes a 32-bit time associated with the first point, and a second 32-bit number that is the time between the two most recent events used for this request, then including a timestamp with each data point that is the time measured from the most recent one of those events would allow for 16-bit timestamps. The host program would need to add each time offset to the time of the first point to get the plotting time value. If this sum is greater than the given event time difference, then subtract this event time difference from the sum to get the time value suitable for plotting the data point. Note that by definition the first time offset would be zero.

Assume that we get an interrupt whenever a clock event is written into a FIFO. In response to the interrupt, sample the 1 MHz free-running timer on the MCchip on the 162 board and save it in an array of 256 entries, one for each clock event. Besides the sampled time of each event, also measure the time between such events. This time between the two most recent events would be used in replies for the plot data described above.

IRM Installation and Configuration

Thu, Dec 30, 1993

Several steps must be taken to bring up a new local-station/IRM based upon the MVME-162 board. These are notes based upon experience in setting up node 561.

Be sure that the J20 jumpers are set to provide backup battery power for the on-board non-volatile memory. The two jumpers should be positioned away from the VMEbus P2 connector.

Make sure that a digital IndustryPack module is installed in slot B. If it is not, the system cannot run successfully.

Via the 162bug command **`pf 0`** and **`pf 1`**, select the baud rate for the 162 board console and host serial ports, respectively, to 19200, or whatever desired.

Via 162bug command **`cnfg;m`**, set the ethernet hardware address to be used. For example, enter 024000000561.

Via 162bug command **`env`**, change several parameters to match those used in another 162-based station. One key item is the Network Auto Boot Configuration Parameters Pointer, which should be set to `FFFC1000`.

Via 162bug command **`niot`**, set network parameters, such as

Node Control Memory Address	`001E0000`
Client IP Address	131.225.123.214
Server IP Address	131.225.123.215
Subnet IP Address Mask	255.255.0.0
Gateway IP Address	131.225.126.200
Boot File Name	`system`
Boot File Load Address	`120000`
Boot File Execution Address	`120000`

The 162bug command **`set`** `mmddyyhhmm` turns on the real time clock. This is needed for timing during the following TFTP transfer.

The 162bug command **`nbh`** downloads the system code. The TFTP protocol is used to transfer the system code known by the filename “system”.

Use the 162bug command **`bf`** `FFE00000:40000 0` in order to clear out all non-volatile memory on the CPU board. This will cause the system, upon initialization, to install a default table directory and initialize several system parameters.

Use the 162bug command **`go`** `120000` to start the system code.

[At this point, the code runs for the first time. It aborts, because it cannot successfully return to 162bug following table directory initialization. Running the code a second time should succeed. The IPARP table will be initialized, as well as the TRING table, used by networking software. The low two bytes of the ethernet address should also appear at 405046 as the local station node#. The **niot** parameters are used to initialize the IPARP table at 40E010 with the station's IP address, subnet mask, gateway IP address, and MTU values. For example,

```
83E1 7BD6 FFFF 0000 IP addr 131.225.123.214, mask
83E1 7EC8 05DC 05DC gateway IP addr, MTU's]
```

Install the 09xx value, for example 09BF, as the UDP node# at 40507E. [The default value of 0964 will already be there, allowing immediate access by a node that can be configured to target the new IP address via node# 0964. This allows ordinary UDP-based Classic protocol to be used via ethernet to access the new node. The sender needs to have the appropriate IP address installed in the corresponding entry of the IP Address Table used by Acnet. But the receiver needs only know its own UDP node#.] Reset the system to let all this work.

Install the Acnet IP address table, which gives the IP addresses for the 09xx nodes. Copy from another station that has this table at 40FA00. (In a 133-based system, it is at 10FA00.) [The pointer to the 512-byte physical node address table used by Acnet should already be installed as address 40F800 at TRING+$78, or 405078.]

[A number of application program names should already be installed in the PAGEP table, as follows:

PAGEMDMP	Memory Dump
PAGEPARM	Parameter page
PAGEEDAD	Edit Analog Descriptors
PAGEEDBD	Edit Binary Descriptors
PAGELAPP	Local appl params
PAGECRTI	Remote page access
PAGEDNLD	Download page
PAGEECHO	Ping, Echo client
PAGENETF	Network frames
PAGEMBLK	pSOS-allocated memory blocks]

Copy any desired programs from another station, say 09BE, via the download page.

For operation away from Fermilab, also copy the HELPLOOP code from node 096F. This is a text "file" that is used to produce the prompting text for local application parameters accessed via page E.

[The data stream used for network diagnostics should already be installed at 401C00. Currently, this looks like:

```
8001 0010 0008 1000  Queue size is $1000.
0010 9000 0000 0000  Queue starts at 109000.
4E45 5446 5241 4D45  NETFRAME
```

Page F is used to display these diagnostics.]

[The default BADDR table should already be installed at 40A000. This provides some dummy non-volatile byte addresses in the 40FFxx area that can used for various system enable bits.]

[A minimum data access table should already be installed at 401000, such as:

```
7F00 0001 0000 0000  Repeat the following at 15 Hz
0000 0000 0000 0000

0405 0000 0040 A000  Update binary status data.
0000 0000 0000 0080  (BADDR at 40A000.)

1D00 0000 0000 0000  Run all enabled local appl's.
0000 0000 0000 0000]
```

Use 162bug command **env** to set network auto-boot parameters so the TFTP transfer can occur automatically at reset time, or perhaps only at power-on reset.

Load and enable local applications LOOPECHO, LOOPTFTP, LOOPAAUX. Use page E to do this, specifying the appropriate parameter values and labeling enable bits via page B.

```
LOOPECHO          (supports UDP Echo testing)
      Enable Bit# 00A9  UDP ECHO ENABLE

LOOPTFTP          (TFTP server)
      Enable Bit# 00B9  TFTP ENABLE

LOOPAAUX          (needed at Fermilab only for Acnet)
      Enable Bit# 00AE  ACNAUX ENABLE
      PNA node# 0921    (Gets PNA table from CNS33.)
```

To keep a copy of the system code in Flash memory, reset and enter 162bug. Then enter the following command:

```
PFLASH 120000:20000 FF880000
```

To recover code later from Flash memory for execution, enter this command:

```
BM FF880000:10000 120000
```

To change the IP address configuration, several changes must be made. Note that if these changes are to be made from 162bug, all `004xxxxx` addresses must be changed to `FFExxxxx`. This is because the non-volatile memory is mapped to the latter address upon reset to 162bug. When the system code initializes, this mapping is changed to `004xxxxx`.

Change the IP address, subnet mask, gateway address via the **niot** command. Then change the corresponding parameters in memory at `0040E010`.

Normally, we turn on the Network Boot option via the 162bug **env** command. When the system is powered up, it uses the TFTP protocol to copy the system code from the given Server IP Address. After the transfer is complete, it automatically begins execution of that code at `120000`. To turn off this option, one must issue a Break when 162bug is starting up. Then one can use **env** to turn it off. For systems installed outside of Fermilab, booting can go faster if the Server is a local node, although TFTP can work successfully across the Internet. To perform the transfer manually, say, to get a copy of the latest system code, use the **nbh** command.

One can maintain a local copy of the system code on a workstation disk. Use the workstation's TFTP client command to do this (in binary mode) from a running IRM system. The system code will be copied onto a disk file. Then use the **niot** command to change the Boot File Name to access the workstation's disk.

IRM Software Overview

Robert Goodwin

Wed, Nov 16, 1994

Introduction

Internet Rack Monitor software is an evolution of that used in several front end control systems at Fermilab and elsewhere. This latest version runs on the MVME162-22 cpu board with MC68040 cpu, 4MB dynamic ram, 0.5MB static ram, 1MB flash memory, ethernet interface, and support for up to four IndustryPack daughter boards. The latter allows connection to I/O signals via ribbon cables to digital and analog interface boards mounted inside the IRM chassis. The ethernet interface allows network connection and supports widely-used Internet protocols that allow data request and setting access as well as alarm reporting, all based upon the UDP (User Datagram Protocol) transport layer.

Local database

The nonvolatile memory houses a number of configuration tables that characterize each station's installation. Included in these tables is a local database for analog channels and digital bits. It includes text and scale factors used by local control applications for scaling as well as alarms reporting. It also houses down loaded code for local and page applications in a memory-resident file system. Local applications are used for closed loop support and for system extensions such as TFTP protocol server support. Page applications support a virtual console access to the system for local control, configuration and diagnostics use. The system code is acquired from a server station by the prom-based 162bug via the TFTP protocol at boot time.

Cyclic data pool activities

IRM software is a collection of tasks that use the pSOS operating system kernel. The primary activities of the system are synchronized by an external timing signal, which may be an external trigger input, or it may be decoded from a "Tevatron clock" signal. (In the absence of a synchronizing signal at 10Hz or 15Hz, the system operates asynchronously at 12.5Hz.) At the beginning of each cycle, the data pool is refreshed according to instructions in the nonvolatile Data Access Table that are interpreted at the start of each cycle. Also, all active local applications are run to operate on the fresh data for closed loop jobs or more complex data pool updates. (A local application is compiled and downloaded separately from the system code itself.) After the data pool is refreshed, all active data requests having replies that are due on the present cycle are fulfilled and delivered to network requesters. Alarm scanning is performed on all analog channels and binary status bits that are enabled for such monitoring. Finally, the currently active page application is run. So, the data pool is accessed by local application activities, replies to data requests, alarm scanning, and the current page application.

Synchronization

The Tevatron clock signal can carry up to 256 events. The IndustryPack digital board decodes each event and interrupts the cpu, allowing time-stamping of each event. Status bits derived from this event activity are part of the data pool and can be used to synchronize data pool updating and local application activities. The time stamps can be applied to data points returned in response to data requests for up-to-1000Hz data available from the IndustryPack analog board; i.e., one can plot 300Hz data, say, with time stamps measured from a selected clock event.

Data request protocols

Due to the evolution of this front end system, three different data request protocols are supported. The original request protocol is called Classic; it is used by stations that communicate among themselves as well as by the Macintosh-based parameter page application developed by Bob Peters. A second protocol was designed by the Fermilab D0 detector people to fulfill their specific needs. The third is that used by the Acnet control system at Fermilab.

Data server logic is included for both Classic and Acnet request protocols. This allows one station (the server) to be targeted by a requesting host with a request for data from other stations, in which the server station forwards the request via multicasting to the other stations and compiles their individual responses into the single reply it delivers to the requesting host. The purpose of this logic is to reduce the number of replies a host might have to endure, in response to a request for data from many different stations. The server station can do this efficiently because of the built-in logic in each station that combines multiple replies—due on the same cycle to the same destination—into a single network datagram. For example, suppose three hosts make requests for data from the same 10 stations at 10Hz, and each host uses the same server node. The server node will receive 10 composite replies each 10Hz cycle, and it will send 3 replies to the three hosts, for a total of 130 frames/second, thus requiring each host to receive only 10 frames/sec. Without the server station, each host would have to receive 100 frames/sec.

Alarm handling

Alarm scanning is performed on all selected analog channels and digital bits each operating cycle, as mentioned above. When a change in alarm state (good-bad or bad-good) is detected, an alarm message is queued to the network to share this news with the outside world. Such alarm messages can be multicast, so that multiple interested hosts can learn of them. A local application can also sample such alarm messages and reformat them to target a designated host alarm server, as is required in the Acnet system. As a local diagnostic, such alarm messages can be encoded for display or printout via the station's serial port, including both locally-generated ones as well as those received by that station listening to the alarm multicast address.

Diagnostics

Several diagnostic features are included in the IRM design. The digital IndustryPack board provides test signals that are driven by interrupt and task activities. The interrupt signals are also displayed on 8 LEDs. These signals can be connected to a logic analyzer to capture timing and related program activities of the station's operation.

A suite of page applications is available to perform various diagnostic displays via the virtual console support.

1: Display areas of memory from any station(s), with 10Hz updating. Clear blocks of memory, or copy them from one station to another.

2: Display Tevatron clock events with 10Hz updating. Capture and display network frame activity, providing a kind of built-in "poor man's sniffer", in which the timing of frame reception or transmission processing is shown to 1 ms resolution within the time-of-day-specified operating cycle.

3: A network client page allows exercising the standard IP ping and UDP echo tests, as well as Network Time Protocol and Domain Name Service queries.

Diagnostic timing of tasks, interrupts

Thu, Apr 7, 1994

On the Internet Rack Monitor is a 26-pin connector of test points that permit monitoring of system software timing. Some of the signals measure the time taken by various tasks, while others measure the time taken by interrupt routines. Here is a layout of the test points connector signals as viewed from the front of the IRM. Tasks are in plain text style, while *interrupt routines* are in italics.

(put picture here)

15. *A/D scan interrupt*. 1Khz from each analog IP board.
14. *Clock Events interrupt* from digital IP board.
13. SNAP task. Handles Internet Protocols IP,ARP,ICMP,IGMP,UDP.
12. RETDAT task. Handles Acnet data request/settings protocol.
11. D0 task. Handles D0 data request/settings protocol.
10. Acnet task. Supports Acnet header-based communications.
9. Serial task. Handler for RS-232 serial input.
8. Server task. Delivers replies to Classic protocol server requests.
7. QMonitor task. Provides cleanup/timeout support for system activities.
6. Update task. Updates data pool, runs local appl's and fulfills active requests.
5. Date/time task. Provides time-of-day support.
4. Small memory dump task. Displays 8-bytes of memory on last line of CRT.
3. Application task. Manages activity of current page application.
2. Console task. Supports local console I/O, real or emulated.
1. Alarms task. Scans for and reports analog, binary and comment alarms.
0. Classic task. Handles Classic data request/settings protocol.

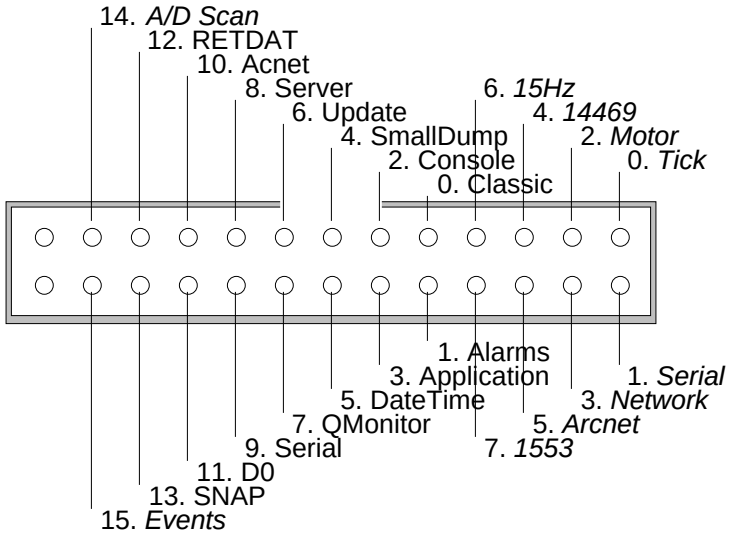
7. *1553 interrupt* for end-of-command from 1553 chip used in D0.
6. *15 Hz interrupt*. Handler for 15 Hz cycle interrupt from external timer.
5. *Arcnet interrupt* from Arcnet chip used in SRMs.
4. *14469 interrupt*. Local console interrupts from crate utility board.
3. *Network interrupt* from token ring or ethernet controller.
2. *Motor interrupt*. Motor step interrupt support at 150 Hz.
1. *Serial interrupt* from SCC serial interface chip.
0. *Tick interrupt* providing 100 Hz timing reference to kernel.

IRM Test Points Connector

Diagnostic timing of tasks, interrupts

Thu, Apr 7, 1994

On the Internet Rack Monitor is a 26-pin connector of test points that permit monitoring of system software timing. Some of the signals measure the time taken by various tasks, while others measure the time taken by interrupt routines. Here is a layout of the test points connector signals as viewed from the front of the IRM. Tasks are in plain text style, while *interrupt routines* are in italics.



15. *A/D scan interrupt*. 1Khz from each analog IP board.
14. *Clock Events interrupt* from digital IP board.
13. SNAP task. Handles Internet Protocols IP,ARP,ICMP,IGMP,UDP.
12. RETDAT task. Handles Acnet data request/settings protocol.
11. D0 task. Handles D0 data request/settings protocol.
10. Acnet task. Supports Acnet header-based communications.
9. Serial task. Handler for RS-232 serial input.
8. Server task. Delivers replies to Classic protocol server requests.
7. QMonitor task. Provides cleanup/timeout support for system activities.
6. Update task. Updates data pool, runs local appl's and fulfills active requests.
5. Date/time task. Provides time-of-day support.
4. Small memory dump task. Displays 8-bytes of memory on last line of CRT.
3. Application task. Manages activity of current page application.
2. Console task. Supports local console I/O, real or emulated.
1. Alarms task. Scans for and reports analog, binary and comment alarms.
0. Classic task. Handles Classic data request/settings protocol.

7. *1553 interrupt* for end-of-command from 1553 chip used in D0.
6. *15 Hz interrupt*. Handler for 15 Hz cycle interrupt from external timer.
5. *Arcnet interrupt* from Arcnet chip used in SRMs.

4. *14469 interrupt*. Local console interrupts from crate utility board.
3. *Network interrupt* from token ring or ethernet controller.
2. *Motor interrupt*. Motor step interrupt support at 150 Hz.
1. *Serial interrupt* from SCC serial interface chip.
0. *Tick interrupt* providing 100 Hz timing reference to kernel.

ROM Boot Configuration

Auto-start for 162-board systems

Wed, Apr 6, 1994

This note describes how to prepare a version of the local station/IRM system code for automatic boot using the ROM Boot option supported by 162Bug.

For most 162-board-based systems, such as IRMs, the normal start-up practice is to use the Network Boot option under 162Bug. After reset, or optionally after power-on reset only, when 162Bug takes control, it can be set up via the **env** parameters to automatically download over the network from a server node using the TFTP protocol. A server node can be any local station/IRM, or it can be a workstation or other host. When it is a local station/IRM, the file name that should be used in that file transfer protocol is *system*. The TFTP local application, which provides such server support, recognizes this file name as a reference for 128K of memory at `00120000`, where the system code is located before it is transferred to `000E0000` for execution. When the server node is another host, the file name should be as is appropriate for that host to indicate the executable image file to be transferred.

For a server node, however, it is desirable that it come up by itself without having to use TFTP, as it is the source for other stations. The 162Bug program includes support for a ROM Boot option, which means that local memory will be searched for a valid program to execute, and control will be passed to it. A valid program must conform to certain rules. A 16-byte header precedes the code, and a checksum word is at the end. This is explained in detail in Chapter 1 of the manual "Debugging Package for Motorola 68K CISC CPUs User's Manual."

These steps can be used to prepare for automatic execution via ROM Boot:

1. In an unused area of DRAM, say `00200000`, prepare a header as follows:

```
00200000    424F  4F54  0000  0010   Ascii 'BOOT', offset to code start
00200008    0002  0000  5359  5354   Length of code, name of code SYST
```

2. Copy the system code into the memory just after this header, at `00200010`.
3. Enter 162Bug and use the **cs** command to compute the checksum word.

```
cs 200000:FFFF
```

4. Using **mm**, install the checksum word at the end of the code, at `0021FFFE`.
5. Program the Flash memory with the result, as follows:

```
pflash 200000:20000 FF880000
```

6. Using the **env** command, insure that the ROM Boot option is enabled and the "ROM Boot Direct Starting Address" `FF880000` is entered.

The above procedure allows a server node to come up automatically after power-on.

Arcnet Gateway

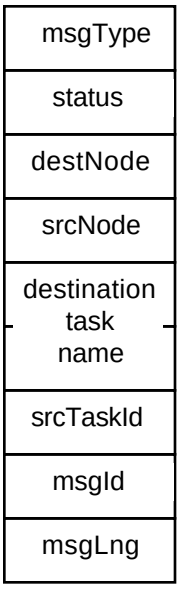
Transport across multiple networks

Feb 26, 1991

The Arcnet network is used to interface the SRM's to a VME Local Station. The Local Stations are networked via token ring. A means of communicating from a token ring node to any Arcnet node can be provided if the local stations provide a gateway service. This note describes a way to achieve gateway for the local stations.

Communication protocols used between the local station and an SRM are based upon use of the Acnet header, a task-to-task communication protocol developed at Fermilab for accelerator control systems. The implementation of the Network Layer in the local station systems uses this header to support task-to-task communications between two nodes on a single network. The essence of this scheme is to use multiple Acnet headers to describe the individual "hops" in communications between two tasks on nodes that span more than a single network. The Acnet header provides a wrapper or envelope to provide for delivery across one hop in the communications path.

As an illustration, take the simple case of a local station (LS1) which wants to display some memory that is located on an Arcnet node (A3) which is attached to another local station (LS2) which is considered to be node A0 on the Arcnet network. For this case, two Acnet headers are sufficient. The Acnet header format is as follows:



LS1 sends a data request message, using the simple protocol used by SRM's, preceded by two Acnet headers. The first (outer) one is the one which specifies the request message type, the destination node LS2, the source node LS1, the

destination task name GATE, the source task id of the original requesting task, the message id assigned by the requesting task, and the message length that includes both Acnet headers (18 bytes each) and the size of the data request message itself. The second (inner) header, which is treated as part of the message contents for the first hop, specifies the same request message type, the destination node A3, the source node A0, the destination task name SRMD, the source task id of the GATE task, the message id assigned by the GATE task, and the message length which is 18 less bytes than the first message length.

For the return path, the reply message from the SRM that includes the memory data that was requested carries a near-copy of the second header described above, with the message type changed to a reply and the status word filled in. The GATE task receives the message because of the source task id. It uses the message id that it assigned with the request message to look up its copy of the first header and precedes that header with the reply message (including its Acnet header) that it just received. It then transmits it according to the source node of the original requester. The original requester receives the entire message including the two Acnet headers. Stripping away the two headers, the requested memory data remains. The status word may be returned copied into the first header (by the GATE task).

This scheme can be easily extended for more hops. The gateway logic becomes no more complicated, as it only handles getting the message across one hop. All of the intermediate destination tasks would presumably be gateway tasks. The user (original requester) must know the whole picture in detail. So, the scheme trades complexity of system software support for the detailed knowledge of the entire transaction that must be known by the user.

For routine maintenance needs of providing access by token ring nodes to data structures in SRM memory, the scheme should be more than adequate. The extendibility of the scheme may help with unknown future needs.

Arcnet Support

Implementation for Local Station

Jun 10, 1991

Introduction

Arcnet is supported by the local station system software to permit access to Smart Rack Monitors (SRMs) which are used in the new Linac controls. As an arcnet node, an SRM interfaces to 64 A/D channels, 16 D/A channels, 8 bytes of binary I/O, and additional I/O depending upon the installation of daughter boards. Onboard an SRM is a 68332 processor, which is used both for data collection and arcnet communication.

The general plan for networking to SRMs is to use Acnet-header-based network messages, which support generic task-task communication over a network. A local station connects to a few (1–8) SRM arcnet nodes. Every 15 Hz cycle, the local station requests all data from each SRM to be collected and returned in a single frame. (In this way, the SRM does not need a 15 Hz interrupt signal to announce the start of a new cycle.) The returned data is mapped into local station analog channel readings and binary status byte readings. Use of a broadcast request permits multiple SRMs to simultaneously prepare their data for arcnet transmission. Each SRM returns a single frame of data containing readings of all types. The token-passing arcnet network hardware arbitrates frame delivery.

SRM-based data acquisition is designed to collect data more efficiently than the previous scheme (used heavily in the D0 system) of using 1553-based Rack Monitors. The overhead of processing a single arcnet frame replaces the overhead involved with processing the many 1553 commands to collect data from the analog and binary hardware interfaces in the D0 Rack Monitors. The inclusion of a processor onboard the SRM allows consideration of implementing certain special handling such as closed loop logic in the SRM rather than in the local station.

Network Layer

As used in the local station, the Network Layer refers to support of Acnet-header-based network messages. Providing task-task communication across a network, it provides a higher level interface for applications that run in the local station. A lower level protocol may have been sufficient for use with SRMs, but the Network Layer permits arcnet communication to take advantage of more software that already exists.

Transmission

The current network routines do not include an argument to specify which network is being used, so the use of Arcnet communications is based upon a certain range of destination node#s. Arcnet node#s are of the form $7Axx. For

reasons having to do with setting support for the SRMs, the range of xx is liimited to $A1–BF.

A message to be sent to a network is placed into an allocated message block, and the routine OUTPQX is called to queue the message to the network. It places the message block pointer into a network output queue. With token ring, this is the OUTPQ system table. With Arcnet, it is a new table statically allocated in on-board ram. (A network output queue need not be in non-volatile memory.) The OUTPQX routine checks the destination node# to determine which queue to use.

When it is time to flush queued messages to the network hardware, the routine NetXmit is called. It now accepts a parameter that indicates which network output queue is to be flushed. (The Network Layer routine NetSend calls NetXmit once for each network.) It concatenates consecutive messages destined for the same node into frames according to the maximum frame size for that network. Most of the NetXmit logic is independent of the network, but a few local variables are set that depend upon the network being used. They are the network# (0 or 1), network board address, network output pointer queue, TPL header, and the maximum frame size. The TPL header is used to emulate the token ring Transmit Parameter List chain that contains references to the spooled frames to be processed by the network hardware. Transmit interrupt processing sequences to the next frame waiting in the TPL chain.

At the end of the NetXmit routine, when a frame is ready to be sent to the hardware a special routine is called that depends on the network being used. This routine ensures that the frame just placed into the TPL chain will be ultimately sent out by the hardware. If the network is not already busy, it forces a transmit interrupt to get it started. If it is busy, the transmit interrupt that results from completion of the current frame will do the job as it follows the TPL chain.

Data structures for arcnet

Several data structures are significant for arcnet communications. The ARCPQ is the arcnet output pointer queue for messages described above. The ARCXMTB is the circular buffer used to hold prepared arcnet frames which are referenced by entries in the ARCTPLH transmit parameter list chain. The ARCRCVB is a circular buffer into which received frames are copied from the hardware buffers. The SRMTABL is used for communications with SRMs specifically, especially for data acquisition. A set of arcnet variables is maintained in the TRING system table, which holds some common data structures with token ring. Note that a special word at TRING+$32 must be set to ‘AR’ to enable a local station’s use of arcnet at all; *without that key, the system will not even look for the arcnet hardware board,* the CC121 VME module made by CompControl.

Data acquisition

Arcnet data acquisition from SRMs is directed by entries placed in the data access table, as is all local station data acquisition. The first of three types of SRM entries broadcasts a cycle request message to all SRMs. Later entries await the arrival of the reply frame from each SRM, and the third type is used to copy the received data into local station analog channel and binary byte readings.

Since all data acquisition is processed while the Update Task is active, in order to preserve correlated data, a special check is made by the arcnet receive interrupt routine for data acquisition replies to direct them to the appropriate message queue. (This is normally done by the ANET Task, which handles acnet-header-based frames and routes each message to the appropriate message queue, but the ANET Task cannot run until the Update Task has finished. Since data acquisition from SRMs is the only reason for supporting arcnet, it was decided that this special case handling was justified.) Data acquisition replies are detected based upon the source task id (in the acnet header) that identifies the network connection used by the Update Task for making such requests.

SRM support

Details of the support for SRM data requests and settings, including the formats for the data access table entries and device setting parameter specifications, are found in the document called “SRM Message Protocols.”

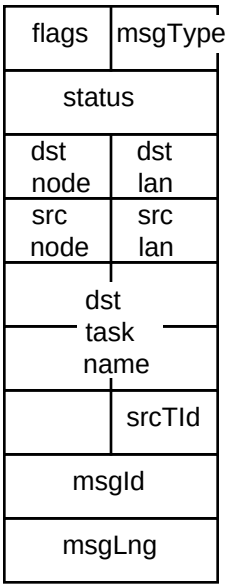
Simple Protocol for SRMs

Protocol #4

Sep 26, 1990

For VME Local Station communications with the SRM arcnet nodes, a choice of protocol must be made. One can use an existing protocol already known to the Local Station, or one can invent a new one designed for the purpose. This option—called “#4” in our informal discussions due to the existence of support already for the Classic, D0 and Accelerator protocols—should be simple, or it would not be worth the effort. An idea for a suitable protocol is explored in this note.

As an aid to get started, assume we use the Acnet header as a basis for a simple protocol design. It is well-known around the accelerator division and provides for expandable and generic task-to-task communications. It allows both one-shot and repetitive replies to generic requests. For reference, its layout is repeated here:

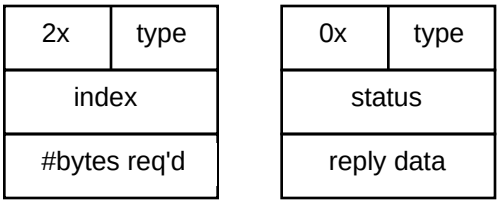


The msgType can be a Request, a Reply, or a USM (unsolicited message). The Request demands a reply. The USM demands *no* reply. The destination task name for a request or USM allows designing a large number of non-interfering protocols, since only the tasks involved in the communication must understand the protocol used in the rest of the message beyond the header. The source taskId provides for routing the reply back to the requester. Multiple requests between tasks are distinguished by the msgId. The msgLng gives the entire message length including the Acnet header (18 bytes) itself. A flag bit in the msgType byte indicates whether a request expects a single reply or multiple replies. Network Layer software supports the use of the Acnet header to provide the task-to-task communications. A task connects to the network to announce its support for handling requests destined for a given destination task name and provides a

message queue that enables it to receive such requests and any replies to its own requests. (Note that a task name is here not the same as a task name known to the operating system kernel.)

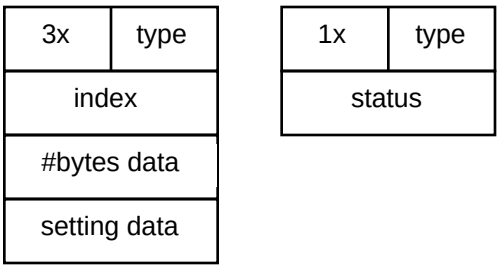
Additional items needed in a simple message protocol for data requests and settings are a message type (beyond the generic msgType mentioned above), a device index and either the #bytes of data requested or the setting data.

Consider the following layout for a data request and reply:



The value of “x” is the length of the index value. This would be 2 for channel or bit numbers and 4 for memory addresses. The type byte can denote analog data, binary status or memory data. The reply can include the same value used in the request.

Consider the following formats for a setting and its acknowledgment:



Again the “x” nibble gives the size of the index value. The #bytes of setting data is included in order to allow grouped settings. Without this consideration, it can be inferred from the msgLng word in the header.

Whether support for this simple protocol is worth the effort is yet to be decided.

SRM / Arcnet Variables Prose

What do they all mean?

Jun 10, 1991

There are diagnostic and operational variables which are part of the local station support of Arcnet and the Smart Rack Monitors (SRMs). The meaning of these variables is described herein. The layout of the variables whose names are used here is shown in the documents “Arcnet Variables” and “SRM Variables.” While a full understanding of these variables requires studying the source code, this note provides a concise summary for diagnostic reference.

Arcnet Variables

The variables for general arcnet driver support are stored in a section of the `TRING` (token ring) table. Besides the following, the use of arcnet at all by the local station must be enabled by setting the word at `TRING+$32` to ‘`AR`’. The source code that uses these variables is the `ARCINT` module.

ArcAddr—the base address of the COM9026 chip that supports arcnet.
D1—Should always be the value $D1, indicating that the arcnet chip successfully reset before the local station system initialized its arcnet logic.
SId—The one-byte local station arcnet node#. Set along with the D1 byte.
bBusy—Bit pattern indicating which hardware receive buffers are full. Not used.
porCt—Count of power-on resets of the arcnet chip. Normally zero.

stat—The last arcnet status byte reading captured at time of arcnet interrupt.
lastDid—The last destination arcnet node# byte used for transmission.
reconfCt—Diagnostic count of arnet network reconfigures that result from an arcnet node being added or removed from the network.
waitMax—Maximum time to wait after an arcnet transmission before timing out. Initialized to 32 (units of 0.5 msec).

lngErrCt—Count of invalid length errors (odd or <4) in received arcnet frames.
idErrCt—Count of errors in destination id of received arcnet frame
auxErr—Auxiliary error status
Invalid destination node# in received frame
Bad length when invalid length (odd or <4) error

rBufOff—Offset to receive hardware buffer (toggles between 0 and $200)
rFrameCt—Longword count of frames received

xBufOff—Offset to transmit hardware buffer (always $400)
xFrameCt—Longword count of frames transmitted

timOut—#times no transmit interrupt within waitMax timeout value

noTokn—#times cannot recover TA after disabling transmitter after timeout.
xFail—#transmissions to invalid node. No acknowledge received.
idFail—node# which did not acknowledge.

thisTPL—ptr to TPL entry whose frame last enabled for transmit
xBFull—Bit pattern for busy hardware transmit buffers
xDsbl—Transmitter is disabled hoping to get TA interrupt.
iMask—Current value of the interrupt mask register (write-only)

noTACt—#times transmitter not busy but TA=0.
lastXCycl—Cycle counter when last frame transmitted.
lastXMS—Relative 0.5 msec count within cycle when last frame transmitted.

FwdPtr—Points to itself. Emulation of token ring receive parameter list.
CStat—Emulation of token ring receive CStat word
fSize—Emulation of token ring frame size received

count— Emulation of token ring maximum space available in receive buffer
buffer—Ptr to current buffer area for next received frame

rBufPtr—Ptr to base of arcnet circular receive buffer area
lngErr—Count of invalid length errors (too small) in received frame.
acfcErr—Count of AC/FC errors in emulated token ring header.

badSap—Invalid SAP# not found in SAP table
uIzSap—Uninitialized SAP value (no queue id) in SAP table
auxErr—Auxiliary error status
    Send_X return error status when sending data acquisition message
    SAP value not in SAP table
    SAP value w/o queue Id in SAP table
    Invalid ACFC word (emulating token ring)
    Bad length when invalid length error
sapQErr—Count of errors returned from Send_X

cStatErr—Last received cStat word in error
rErrCt—Count of bad received cStat words.

SRM Variables

These variables are used for SRM communication using data access table entries and setting parameters. The source code that maintains these variables is found in the SRMREQ module.

SRMD tid—The taskId returned from NetCnct for SRMD used by data acq.
SRMS tid—The taskId returned from NetCnct for SRMS used by settings.

SRM ‘up’ status—longword of status bits for each SRM returning data acq.

Create_X stat—Status return when creating message queue.
NetCnct stat—Status return when connecting to network.
SRMQueue stat—Status return from SRMQueue for data acquisition.

SRMS queue id—SRMS message queue id used to call NetCnct.
NetCheck stat—Status return from NetCheck call.
SRMQueue stat—Status return from SRMQueue for settings.

Setting message buffer—Buffer used for last setting “#4” header to an SRM.
Setting data—First 10 bytes of last setting data sent to SRM via SRMSet.

Request message buffer—Last request message sent by data acquisition.

SRMD queue id—SRMD message queue id used to call NetCnct.
Cycle counter—Local station cycle counter of last SRM data acq request.

mRsvd—pSOS message queue buffer (6 longwords)
mHome—(pSOS special case. not used)
mSize—(size of message received)
msgCntOff—(Offset to frame message count word)
srcOff—(Offset to frame source address)
destOff—(Offset to frame destination address)
message ptr—(Ptr to message in frame buffer, base of offsets above.)

msgPtr—Ptr to message in frame buffer received from SRM A1.
msgSize—Size of message received from SRM A1.
msgTime—Relative time in cycle of last message received from SRM A1.
msgStat—Status word from acnet header of last message received from SRM A1.
nRecvCyc—#messages received this cycle from SRM A1.
nRevTot—Total #messages (longword) received from SRM A1.
—above 6 variables repeated for each SRM A2–BF.

SRM Message Protocols

From Local Station over Arcnet

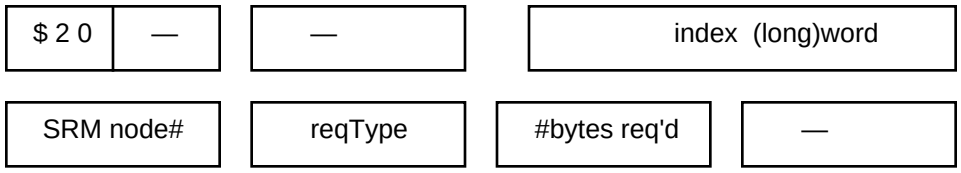
Jul 23, 1991

The new Linac control system local stations support an Arcnet interface to Smart Rack Monitors (SRMs). Each station connects to several SRMs using a private Arcnet network. The SRM acts as a data concentrator to make data access to the hardware more uniform. In the future it may support closed loop algorithms. To the local station, an SRM is treated partly as a data interface and partly as a network node. The simple message protocol is based upon the Acnet header used for task-task commun ications in Fermilab accelerator systems.

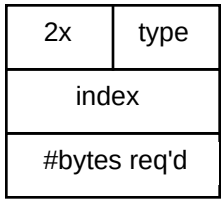
Data interface

As a data interface, three types of entries in the Data Access Table (DAT) of the local station reference the SRM for collecting readings and settings. The advantage that the SRM provides is that a single network frame can deliver all the SRM data in response to a data acquisition request, thereby making access to the data more efficient. The following describes some details of this connection.

Every 15 Hz cycle, the DAT is interpreted by the local station. This *first* type of DAT entry for SRM data acquisition is used to send a request message to the `SRMD` destination task in the SRM. The entry format is as follows:



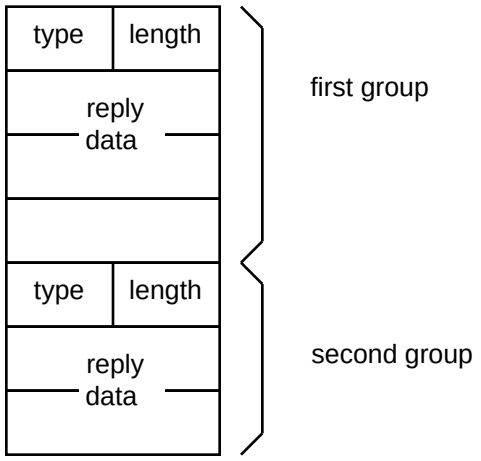
A one-shot data cycle request message is queued to be sent to an Arcnet node or broadcast to all Arcnet nodes. `NetXmit` is called to flush the network queue after this entry. Beyond the Acnet header the format of the message is:



For this case, x is 2, indicating one index word value present. The maximum number of bytes requested is 490 bytes. The type byte indicates the usual 15 Hz data collection. The index word is not used in this case. In response to this message, the `SRMD` task collects all its data using its own version of a data access table, builds a frame buffer with its response data, and returns the response frame to the local station. Type byte values defined for SRM data requests are:

Type	*Function*
01	Request to read and return 15 Hz cycle data
02	Memory dump (index = 4-byte address)
03	Table access (index = table#)

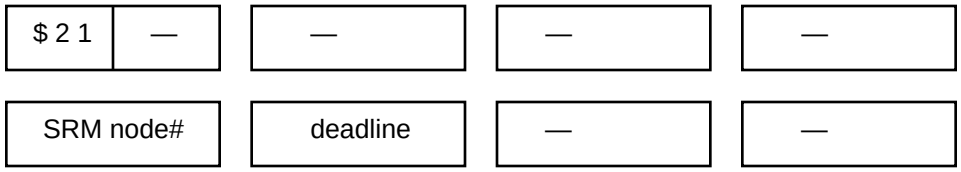
The data response message from the SRM has the following format:



Several groups of variable lengths follow the acnet header. Each group represents data of a different type. The group types,which are SRM table#s, are listed below. The length byte is the number of *words* in the group, including the type/length word. Error status is given by the acnet header status word. The reply message remains in the circular receive buffer until needed by post-processing logic that maps these sections of data into local station analog channels and binary data bytes.

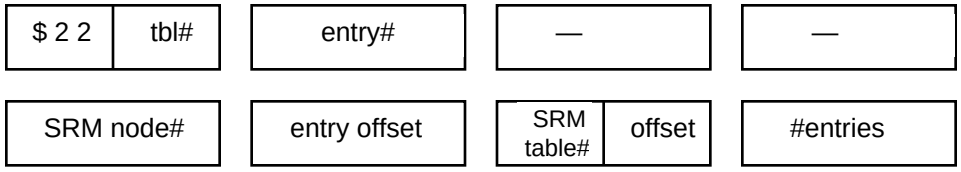
SRM table#	*Data referenced*	*#entries*
01	Linac mpx A/D via daughter-board I/O	16*n words
02	Linac mpx D/A via daughter-board I/O	16*n words
03	spare —	
04	A/D on SRM board	64 words
05	D/A on SRM board	16 words
06	Digital I/O on SRM board	8 bytes
07	Mpx digital input via daughter-board	16 bytes
08	Digital I/O on a daughter board	4 bytes
09	Actel timer board	8 words
0A	Opto-22 digital I/O bytes	2–3 bytes
0B	Digital I/O board #1	9 bytes
0C	Digital I/O board #2	9 bytes

Returning to the DAT entries in the local station, when no more work can be done until the SRM response data has been collected, a *second* type of entry in the data access table is used to wait for the SRM data response.



This is done by waiting on the message queue containing the reference to that response message. When the message from the given SRM is received, or if a deadline time (in 0.5 msec units from the start of the current 15 Hz cycle) is exceeded, processing of the DAT continues. If messages are received from other SRMs besides the one specified in this entry, a record of it is kept so that the succeeding wait type entry that refers to it is immediately satisfied. To insure that old messages are not left over from the previous cycle, the message queue is emptied prior to send the SRM data request message.

A *third* type of DAT entry is used to map the response data into analog channels and binary bytes of local station data.



The *entry#* identifies a target analog channel, for example. (In the usual case, the *tbl#* would be 0, indicating the ADATA table. One can target the setting word of an analog channel, rather than the usual reading word, by using a value of 2 for the *entry offset*.) Also specified is an SRM node# and an SRM table#, given in the table above. A search is made through the structure in the cycle response message for a match on this group type, and data words are copied into successive entries in the ADATA table. If the *#entries* is not equal to the amount of data included of the given type, then only the smaller amount of data is copied. This allows modifications to the SRM's data access table without simultaneously having to change the local station DAT. For the case of digital byte data, the #entries refers to a #bytes; otherwise, it is a #words. Many such DAT entries may be required to map the different groups of data included in the cycle data response frame into the local station's data pool. It is assumed there is only one occurrence of a given SRM table# in the received message.

If an SRM does not respond to the cycle data request by the given deadline, the target channel or byte readings are written as zeros, in order to avoid reporting stale data readings. Also, settings are not accepted for such an SRM. A longword of status bits is maintained that can be assigned as digital data bytes and included in the alarm scan to provide alarms about SRMs which are down.

Examples of DAT entries for SRM data acquisition

Request cycle data from all SRMs:

```
2000 0000 0000    0000
7A00 2201 01F0    0000
```

Wait for cycle data from SRM A2

```
2100 0000 0000    0000
7AA2 0030 0000    0000
```

Map 64 channels of SRM A / D into analog channels 0080–00BF:

```
2200 0080 0000    0000
7AA2 0000 0400    0040
```

Map 16 channels of SRM D / A into settings of channels 0080–008F:

```
2200 0080 0000    0000
7AA2 0002 0500    0010
```

Map 6 bytes of SRM digital data into binary bytes 000C–0011:

```
2205 000C 0000    0000
7AA2 0000 0600    0006
```

Network interface

To support Arcnet in the local station, which has extensive support for the token ring network, a range of node numbers is used for Arcnet nodes. A node number is a word, so there is plenty of addressing space available to do this. The Arcnet range is $7Axx, where xx is the one-byte node# used by the Arcnet interface. Support for motors, described later in the settings section of this document, further restricts the range used for SRM Arcnet nodes to A1–BF, which still permits up to 31 SRM nodes to be installed on Arcnet.

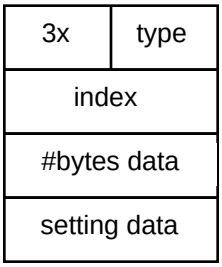
Messages to be sent to an SRM on Arcnet pass through a separate Output Pointer Queue (OUTPQ) than that used for token ring transmissions. (This permits Arcnet activity even during the time that a station is opening onto the token ring network.) The NetXmit code combines messages into frames using the appropriate transmit buffer. There is a separate transmit parameter list chain, which is a queue of pointers to network frames ready to be transmitted on Arcnet. When the frame is ready, a check is made for current Arcnet transmit activity. If it is idle, then an Arcnet transmit interrupt is forced by enabling the TA interrupt. (If it is busy, nothing is done, as the pending Arcnet transmit interrupt will take care of it.) The interrupt routine does error checking on the last frame transmitted, if any, and copies the next frame in the queue to the hardware buffer. It then enables the buffer to hand it over to the Arcnet hardware. In this way, many Arcnet frames can be queued awaiting transmission.

When a frame is received from Arcnet, the interrupt routine copies the frame from the hardware buffer into a circular receive frame buffer. This buffer is separate from the one that receives token ring frames because the token ring chipset has DMA control over its frame buffer. A second hardware buffer is re-enabled to provide a place to receive the next frame. The software emulates the token ring format in the circular buffer. It also checks the Acnet header of each message within the frame and sends a pSOS message reference (consisting of four longwords that include a pointer to the message itself in the circular buffer) to the message queue used by the receiving task, according to an entry in the NetConnect table (`NETCT`). That entry# is given by the `srcTaskId` in the Acnet header for replies. An optional event can also be sent to a task in case the task waits on events instead of the message queue. When a task receives such a reference message about an Arcnet message, it looks exactly the same as if it came from the token ring network. All subsequent processing is unaffected.

To a user program, Arcnet nodes can be accessed in the same way as for token ring. The only difference is that the protocol of the messages supported is different. The SRMs do not support the Classic protocol, the D0 protocol or the accelerator protocol. They only support their own Acnet-header-based protocol.

Settings

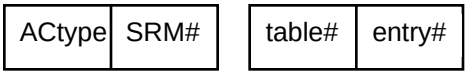
Settings are of several types, including memory, analog D/A or motors, and binary bit or byte control. Each results in a short message sent over Arcnet to the SRM. The message format (beyond the Acnet header) is:



The x indicates the size of the index field. It is 4 for the memory case since the index field is a 32-bit address. It is 2 for the other cases. The type byte denotes a control type# to the SRM using the following values:

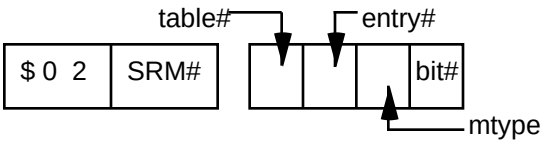
Type#	*Hardware control*	*Index word*	*Data*
05	Motor control	table,entry/mType,bit	#steps
06	Bit control	table,entry/0,bit	dcType/pulse
07	Byte control	table/entry	Data byte
08	D/A control	table/entry	Setting value
09	Memory write	32-bit memory address	Memory data

(The index field syntax above uses "," to separate nibbles and "/" to separate bytes.) In the local station, the index parameter of the message originates in various tables. First consider the analog control case. For a D/A or motor, the analog control field of an analog channel descriptor is required to contain the needed information to effect the setting.



For a D/A analog control field entry, the *SRM#* is the Arcnet node# byte. The SRM *table#* and *entry#* make up the last two bytes.

For a motor, the Analog control type# is constrained to be 02, so there is less space available for the needed parameters.

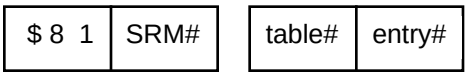


The table# and entry# (byte#) is limited to 4 bits each. The bit# (range 0–7) is the bit# of the byte that contains the pair of control bits that drive the motor. The mType specifies the type of motor interface as follows:

mType Motor interface	*bit# indicates:*
1cw, ccw pulse pair	cw bit
2pulse, direction	pulse bit

For digital control, the BADDR table in the local station contains longword entries for each byte of binary data. The entry is normally a pointer to the source of the byte of data, except for special cases. For the case of an entry in the range $80xxxxxx, the lower three bytes are a 24-bit pointer to a byte in a 1553 command block that accesses one word of digital data. For control, the corresponding command block is found located 16 bytes from the reading command block.

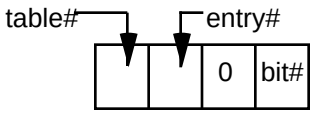
The range $81xxxxxx is used to indicate SRM digital data. The second byte is the SRM node#, and the last two bytes are parameters of the digital interface.



For this case, it is not necessary to contain a pointer to the data byte reading, since the post-processing that is done with the third type of DAT entries above takes care of updating these bytes that are stored in the BBYTE table of raw status data. The processing for the $04 DAT entry skips to the next BADDR entry if it

finds an entry in the range $81xxxxxx.

For digital *byte* control, the index word in the message sent to the SRM is just the second word above. For digital *bit* control, however, the message passed to the SRM has a modified index word, in order to specify the bit# within the byte that is to be controlled. The modified format is:



The table# and entry# are squeezed into a nibble, and the bit# of the byte to be controlled is placed in the low 3 bits.

For analog and binary settings, the SRM keeps a copy of the last values that were sent to the hardware interfaces. When an SRM resets, it re-sends all of these values to the hardware, since it may be resetting from a power-off condition, and the hardware interfaces would have lost their settings.

The local station also re-sends settings for the same reason after a reset of the local station system. For the SRM devices, this means that the local station values are passed to the SRM via Arcnet. The local station values are expected to be correct, since they are updated when a setting message is queued for the SRM and also when the SRM returns setting values in the cycle data response message. These saved values are returned in response to a host's request for setting data.

As an example, when a setting is received for an analog channel which is attached to an SRM, the value is placed in a setting message that is queued for Arcnet. If the queuing is successful, including a check on whether the SRM is actively participating in the data cycle request activity, the setting word in the ADATA table is updated. When the SRM executes the setting, and there are no errors, then it updates its own setting value. If the SRM is unsuccessful in making the setting, this value is not updated. In any case, the following cycle's data response message contains the last value kept by the SRM. In the case that the setting was unsuccessful, this value will replace the optimistic value stored by the local station. In this way, the local station value is nearly always correct. By having the local station store the optimistic value, one can maintain sensible support for multiple user control of the same analog channel, immediate read-back of the setting value, and multiple setting commands that reference different bits in the same byte.

Analog Descriptor Add-ons

Remembering special tidbits

Tue, Feb 25, 1997

When it is necessary to keep special information in the local database that relates to only a few channels, a means of storing it would be desirable. This note explores possibilities for this.

Using the name table

The name table is designed to provide a quick way to search for info according to a unique key. It has been used so far in name searching, for both 6-character names used locally in the Classic protocol and for 16-character names used in the D0 detector control system. The scheme depends upon the use of unique keys. Multiple types of keys may be used, because in each case a type code is specified along with the name. The "name," or whatever unique key is used, must reside within a system table of some kind. The data that is stored in the "name table" is only the entry number of the system table that is referenced by the type code. For 6-character names, for example, the data that is stored in the name table entry is the channel#, along with a pointer to the name as it resides in a field of the analog descriptor table.

In this case, the "name" might be a channel#, as those are unique within a system by definition. A system table would have to exist to hold the information that is special for a few channels. The data word in the name table entry would be the entry number of that new system table. The pointer would have to point to the channel# that must be stored in that entry. So, given a channel#, one can quickly come up with the corresponding entry# in the new system table, if there is one.

A new listype# could be defined, along with new read routines and set routines, so that read/write access to the special information can be supported. Some means of deleting the extra info, and thus opening up a spare slot in the new table, must also be provided.

Reference from descriptor

The spare byte in the analog descriptor could be used to index into an up-to-256-entry table that could contain the extra info. It is quick to find the special info, given a channel#, but one is limited to 256 entries.

Increase size of descriptor

Although this is the easiest path, it requires more memory than may be convenient to relinquish. The 162 boards have only 512K of non-volatile memory. For 1024 channels, we use 64K of this memory for the current descriptor size. With 192K of downloaded program memory, only 256K remains. If the #channels is limited to 1024, then we could double the size of a descriptor entry with no trouble. Not having the special D0 tables saves another 64K. Not expanding the ADATA and BDESC tables

In summary, limiting support to 1K channels and 1K bits, with 128 bytes/analog descriptor table entry, we have:

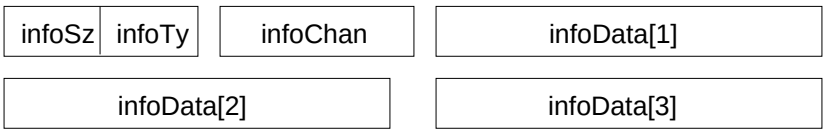
Download program area	192K
ADESC	128K
ADATA	16K
BDESC	16K
Miscellaneous	64K
spare	96K
Total	512K

One could double the size of an ADATA and/or BDESC entry and still have 64K available.

For the case of 133-based systems, which are all those in Linac, the situation is different, although there is 1024K of non-volatile memory available. With the ADESC table set at 150000, and with 2K channels of 128 bytes each, we would extend the ADESC table to 190000, where the binary tables begin.

New CINFO system table

Another implementation of channel-related extra information uses another system table, #25. Each entry contains the channel# key. An entry could be variable length and of different types. Here is a proposed layout:



Each entry is composed of a size byte, a type byte, a channel# word, and arbitrary information. An entry's size must be a multiple of 8 bytes. The example above shows an entry with a size of 16 bytes.

As an example of the use of such an auxiliary table, consider the need for information about channels that have swift digitizer support in Booster IRMs. Up to 8 channels can have such support, but which channels they are is arbitrary. The FTPMAN support must, given a channel#, determine whether or not swift digitizer support exists for that channel. The CINFO scheme can do this. A new library routine can assist in searching this small table.

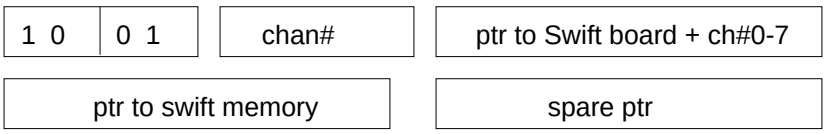
```
Function CINFOEntry(typ, chan: Integer): CINFOPtr;
```

In the case that there is no CINFO entry available for the given channel, NIL is returned. Let the type parameter be 1 for swift digitizer information. When FTPMAN gets a request for swift digitizer data for a given channel, this routine can answer the

question of whether that channel is connected to a swift digitizer channel. For example, the CINFO entry might have this form:

The ptr to the swift digitizer board gives access to the registers for controlling the digitizer. The low 3 bits can contain which of the 8 possible digitizer channels is used. To find the memory occupied by the resulting digitized waveform, one must read a register from the IPIC chip, whose address is fixed, on the 162 board. The register that must be read depends upon which of the four IP board sockets is used. The base addresses of the IP board sockets on the 162 cpu board are FFF58x00, where x ranges from 0–3 for board sockets a, b, c, d. So the board socket index can be obtained from the base address that is contained in the table, such as FFF5820y for board socket 2, or b, with y in the range 0–7 to signify which digitizer channel is used.

In case an separate IP carrier board is used, and the IPIC chip is not used, more information is needed to find the memory address of the waveform. To cover this case, the size should be larger than 8 bytes. Here is a possible layout:



Here is a routine that can return the ptr to a swift digitizer waveform memory:

```
Function SwiftMem(infoP: CINFOPtr): SwiftMPtr;
```

Given a pointer to the CINFO table entry that was returned by CINFOEntry, SwiftMem uses the previous logic to find the base address for the given waveform's memory. It checks for both the short and long cases. It returns NIL if an error.

One can also, in a similar way, design a scheme to help with quick digitizer support.

1553 Control for Co-processors

Keeping control of the interface

Feb 2, 1989

In the Loma Linda VME system, co-processor boards are used to provide waveform generation to drive ramped power supplies. The interface to the supplies is via 1553. Access to 1553 cannot be shared, as sending a new command will cancel one which is in progress. One could make use of semaphores to support resource ownership, but the ramp generation is extremely real-time, with up to 4 power supplies driven simultaneously at a rate of 720 Hz. There is very little time that the 1553 interface is not busy.

The ramp co-processor routinely plays out the ramp and reads back the data from the 4 power supplies at 720 Hz. But digital control needs to be handled somehow. We must be able to turn power supplies on or off, for example. The request to do this will come from the main VME cpu either from its own current application—especially the parameter page—or from a network setting request. The hardware connections are there to allow the VME cpu to talk to the 1553 controller, but it dare not in order to keep from affecting the ramp adversely. We must get the co-processor to do it when it has a brief period of time available to sandwich it in with its other I/O.

The natural way to let the co-processor know about a digital control request is via a message queue in shared memory. By using a queue, several control messages can be awaiting co-processor service. (At the 720 Hz rate, there may not be need for a queue to hold a large number of messages.) There is a system table which contains pointers to co-processor queues. One of these is a co-processor command queue—a separate one for each co-processor.

We can send a message via a co-processor's command queue requesting that a 1553 control action be taken. The co-processor monitors the command queue when it has time available and processes requests it finds there. In the case of the ramp co-processor, the 1553 control must be handled by its interrupt code, which is used to drive the ramp I/O. So, the ramp task level processing may need to use another queuing mechanism to pass such requests to the interrupt code. Or, it may simply use a buffer for the purpose—a one element queue, if you will.

Returning to the VME main cpu, how shall it know to send a message to the co-processor to handle the 1553 control? (The case of supporting digital control is not simple, as has been covered in a separate note called "Digital Control Pulse Delays.") This note describes how to handle this at a low enough level so that the complexities of the higher levels remain unaffected.

Let the various device tables in the VME system be built as if the VME cpu itself will control the 1553 interface. But, at the last moment, when the 1553 transaction is about to be processed, let the code realize that the co-processor must be sent a request message instead to do the 1553 I/O.

Whatever processing goes on from the point of a user pressing the keyboard interrupt key to initiate an off command, for example, ultimately results in a word of digital control data being sent to the 1553 interface. At that point, the routine called `OUTW1553` is invoked. The arguments passed to this routine are a pointer to the command block in 1553 memory where the command word is stored, the data word to be output, and a try count in case of errors. The command block pointer is enough to identify the 1553 controller being accessed. (Each 1553 VME board houses two controllers.) Normally, the job to be done is to copy the data word into the command block and alert the 1553 interface chip to process the command. But, if this command block is one which should be handled by a co-processor, then we build a short message instead and send it to the command queue for the proper co-processor.

How shall we determine whether a given command block should be handled by another cpu? One plan is to key on the address of the command block itself. In the case of the ramp co-processors, each uses a 1553 controller whose base address is $00Ex0000, where “x” is the co-processor number. (Each 1553 controller uses 64K of address space.) This scheme is straightforward if the main cpu assumes knowledge of this formulation. One disadvantage it might have is that the ownership of a given 1553 controller is not program controlled. It depends upon the setting of the address switches on the 1553 board. (One could still get program control by changing the formulation dynamically.)

Another way to detect whether the 1553 control should be passed on is by examining the content of the command block itself. The hi byte of the first word is used to specify an optional offset to a “diagnostics block” for making a record of errors and usage counts. If the byte is zero, no diagnostics are recorded; if it is positive, it is taken as an offset from the start of the command block to an 8-byte area used for placing the diagnostics information. If the byte were negative, it could signify that a co-processor is to do the control of the 1553 controller which houses the command block. Specifically, if the first byte were in the range `$C0–CF`, it could mean that the corresponding co-processor in the range 0–15 should be passed a message via its command queue. A possible disadvantage of this scheme would be that every 1553 command block in that controller’s memory would have to be identified with the proper byte value in order to insure that the main VME cpu would not try to drive that 1553 controller itself.

A third approach is to keep a table of 1553 controller ownership. The table could be indexed by 1553 controller number, which ranges from 0–15 and is obtained from a command block memory address by isolating bits 19–16 of the address. In the version of the system software which uses 1553 interrupts, there is a table of 1553 queue pointers. An entry is placed into this table the first time 1553 I/O is done to a particular controller. One could add an additional field to the entries in this non-volatile table which would declare co-processor ownership for a given controller.

Alternatively, one could place a special code word in the controller's memory that could be interpreted as a declaration of ownership. Let a word near the end of a controller's 64K block of memory space be used for this code. The last two words (at offset $FFFC from the base address) are actually the controller's register block. If we back up two more words before the register block, using the offset $FFF8, we could use the word at that offset for the purpose. To make it definite, the word at the offset of $FFF8 will be the code word. If its value is in the range $C000-C00F, it denotes corresponding co-processor ownership. If the VME system encounters values in this range when about to do 1553 I/O, it should not do the 1553 I/O. If it is sending a word to the 1553, it can instead build a small message to pass to the co-processor via its command queue.

After consideration of the above choices, the last one was chosen. The Data Access Table entries which drive 1553 data acquisition are ignored when the code showing co-processor ownership is encountered. When a word of data is to be sent to a 1553 RT—to set a D/A, for example—a message is built and sent to the co-processor's command queue. When the 1553 Test Page is being run, the code word in memory will have to be altered in order to get it to run. The 1553 driver that is called by that test page will not send any 1553 commands if it finds that the code word exhibits 1553 controller co-processor ownership.

Co-processor Message Queues

How to talk to the VME system processor

Feb 7, 1989

The VME system software supports the connection of co-processors, which are additional cpu boards which are accessible from the VMEbus. Up to 15 co-processors may be supported. There is a system table which contains the parameters specific to each co-processor.

There is a simple message queue communications scheme that allows sending messages to a co-processor and accepting messages from one. A separate queue is used for each direction of communication. This note describes how a co-processor might find the message queues and use them.

The base location of the VME system table directory is at $00100000 on the VMEbus. The table directory contains an 8-byte entry for each system table numbered from 0–30. The Co-processor queue pointer table is table #15. As a result, the 8 bytes for this table at address $00100078 are found as follows:

word	(#entries)
word	(#bytes/entry)
longword	(Ptr to co-proc queue ptr table)

Using the co-processor#, perhaps obtained from some bits of the module status register set by switches on the 133A cpu board, find the corresponding entry in this table. Simply use as an offset the product of the co-proc# and the #bytes/entry word above. There will be found the following:

longword	(Ptr to co-processor command queue header)
word	(Size of queue)
word	(n.u.)
longword	(Ptr to co-processor readback queue header)
word	(Size of queue)
word	(n.u.)

The command queue is used to send messages *to* the co-processor, while the readback queue is used to receive messages *from* the co-processor.

At VME system reset time, the command queue is created by the VME system processor. Its 16-byte header's format is as follows:

word	IN	Offset to next message to go IN to queue
word	OUT	Offset to next message to be taken OUT of queue
word	LIMIT	Total size of queue including this header
word	START	Offset to first entry in queue (after header)
word	KEY	Key initialized to 'MZ' at reset time
byte	INErr	Diagnostic error count by IN user
byte	OUTErr	Diagnostic error count by OUT user
word	INCnt	Diagnostic count of messages placed INto queue
word	OUTCnt	Diagnostic count of messages taken OUT of queue

All offset words are offsets from the beginning of the header. At reset time, the KEY word is set to zero, while the queue is cleared and the header initialized. The IN, OUT, and START words are initialized to 16, the size of the header; thus, the queue contents immediately follow the header. The KEY word is set to 'MZ' as the last act of initialization.

The co-processor examines the command queue header when convenient. When it finds the value 'MZ' in the KEY word, it alters that word to 'MQ' to signal that it has recognized the command queue and will be monitoring it for messages. The VME system will not place any messages into the command queue unless it reads the value 'MQ' in the KEY word.

The diagnostic fields are there to allow inspection of queue activity by those interested.

When a message is placed into the queue, the IN offset is used to determine where to place it in the queue. There must be enough room so that the message can be stored as a single contiguous block in the queue, or it cannot be placed into the queue. The IN user must check the OUT and LIMIT offsets to be sure that there is available space for the message. If the IN offset is ≥ the OUT offset, and there is not enough room to place the message to end at least one word *before* LIMIT, then a word of zero is written at the IN offset, and the IN offset is reset to the START offset value. Then a further check must be made to see that the message will fit before the OUT offset.

After the message has been copied into place, the IN offset is advanced by the length of the message. Note that a message that causes the IN offset to be advanced to *equal* the OUT offset must not be entered into the queue, as the IN=OUT condition signifies an *empty* queue to the OUT user.

The format of the message is arbitrary, but the first word must be the size

word; it gives the length of the message in bytes *including the size word*. In most cases, the message size will be even; in any case, the IN and OUT offsets will always be even.

There are a few formats of messages that the VME system itself can generate. To allow for these cases, the second word of a message is the type word, which denotes the message type. The type word values used in such message will always be less than 256. Private communication messages can be sent to a co-processor by using type word values ≥ 256 and not conflict with those which the VME system may generate.

The following messages are sent by the VME system to a co-processor:

Analog Control

word	size =8
word	type =0–15, 128–143
word	index (any value)
word	data

This is used for making a "D/A setting" to a co-processor. The type value as well as the index value is stored in the analog control field of the analog descriptor for the channel. Bits 6–4 of the type word are used to specify the co-processor # used for determining which command queue is to receive the setting message. The remaining bits of the type word and the entire index word are arbitrary and have meaning only to the co-processor. They are designed to enable the co-processor to determine what device or pseudo-device it should set.

1553 Control

word	size (=10,12)
word	type (=15)
long	ptr to 1553 command block
word/long	setting data (2,4 bytes)

This is currently used for handling 1553 digital control. At a very low level, a decision is made *not* to handle the 1553 output *directly*, but instead to pass a message to a co-processor to handle it. The ramp co-processors in the Loma Linda accelerator system must have exclusive access to the 1553 controller hardware interface that controls the ramped power supplies. This scheme permits digital control to be handed over to the ramp cpu for execution when it has time during its 720 Hz activity.

In the future, one can expect that it will be possible to send a digital data word to a binary control word, denoted by type and index values analogous to those

used for analog control.

Generic control

word	size
word	type ≥ 256
(any)	

Listype #40 is used to send this form of message to a co-processor. The ident, in the long form, is as follows:

word	lan-node
word	co-processor# in range 0-15

It is the same as a channel ident, except that a co-processor number is used instead of a channel number.

D0 Data Requests / Settings

System Implementation

Mar 4, 1990

Introduction

The new message formats for D0 data requests / settings, described in the document "D0 CDAQ Network Data Transmission Protocol" by Alan Jonckheere, use the Acnet header designed by Charlie Briegel to support generalized task-task communications across a network. The Network Layer software in the VME Local Stations supports these Acnet header-based messages. This note describes the implementation of the support for the new data request and setting messages.

Message flow

When a request or setting message is received, it is directed to a well-known taskname RPYR. At initialization, the DZero Request Task creates a message queue (called DREQ) that is used to receive Acnet header-based messages directed to the taskname RPYR. NetCnct registers this taskname to the Network Layer.

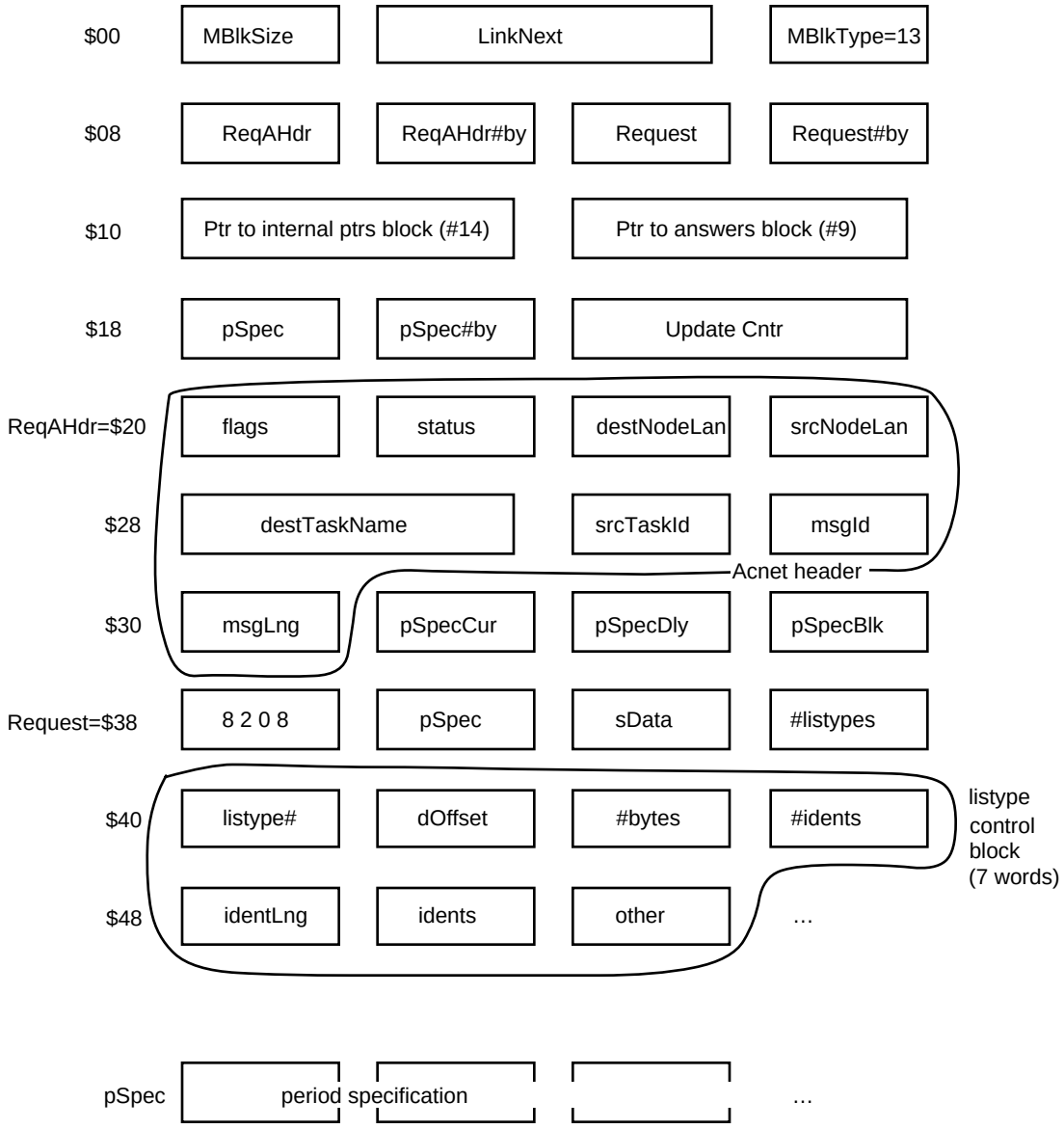
```
Function NetCnct (taskName, queueId, eventMask, VAR taskId);
```

The eventMask is left zero, as the Request Task will simply wait on the message queue rather than wait on an event. The Request Task then enters an infinite loop that calls NetCheck to wait for a message and, upon receiving one, process it.

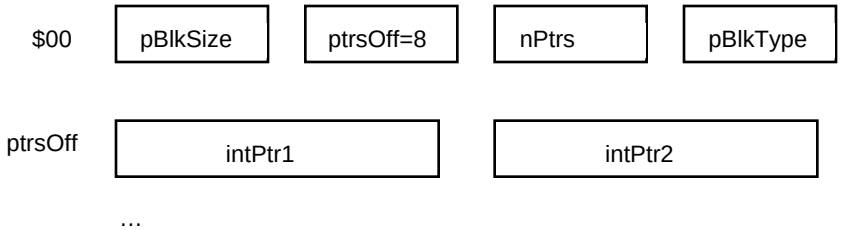
```
Function NetCheck (taskId, timeOut, VAR msgRef);
```

When the function returns with valid status, the message type is checked as found in the first word of the Acnet header. If it is a USM (unsolicited message) with the CAN bit set, the request identified by the message id is cancelled. If it is a request message type, the message following the header (and the format block) is checked. If it is a setting, it is processed immediately. If it specifies a request for data, then a set of 3 message blocks are allocated for support of the new request. (If the request specifies an existing active message id, then the existing request is cancelled.) The basic request block houses the various parameters needed to monitor the request activity. Two pointers are included in that block that point to the other related allocated blocks—the internal ptrs block and the answers block.

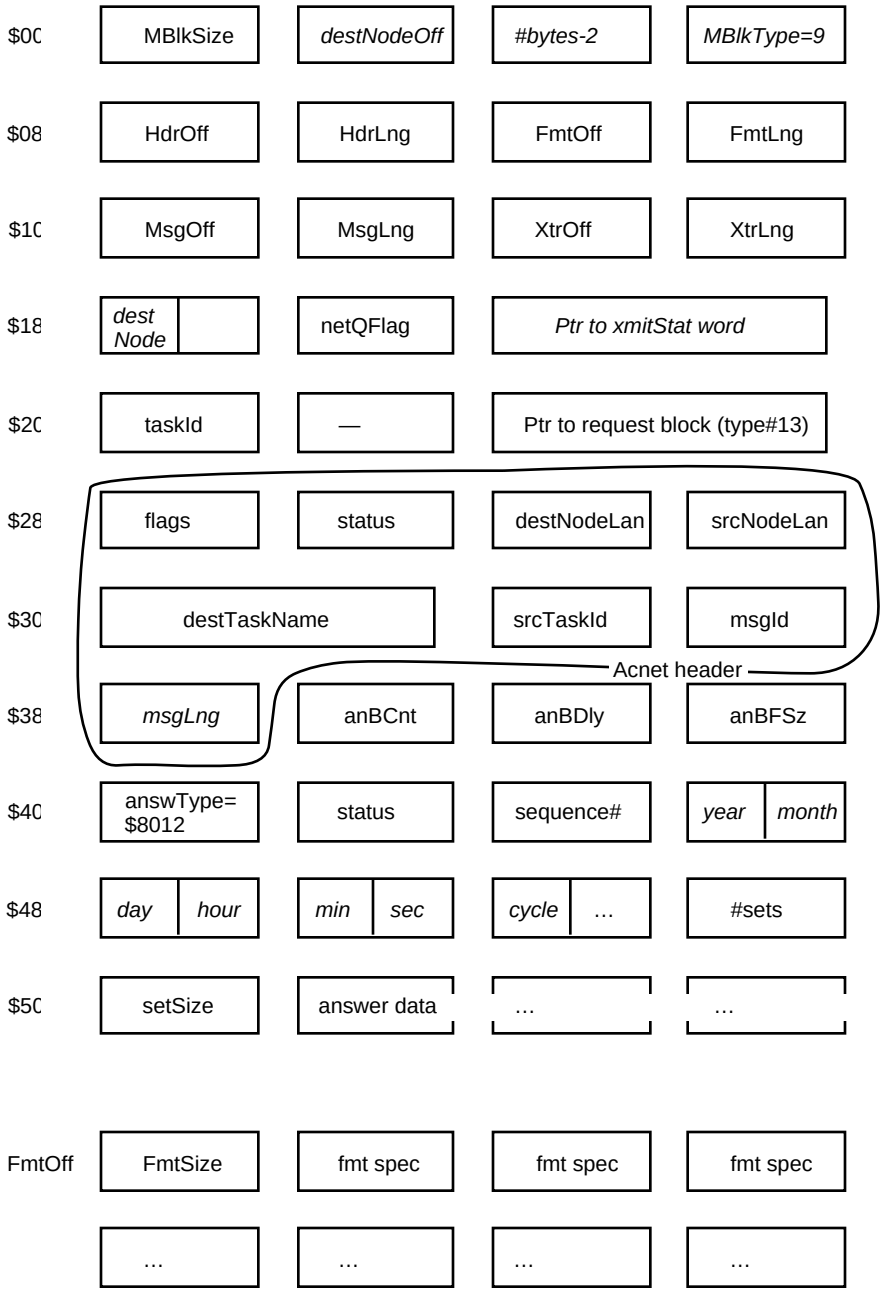
The basic DZero request block (type #13) contains the array of listype control blocks (LCBs) and the period specification.



The Internal Ptrs block (type #14) contains the array of internal ptrs that are used to update the request (build the answers) efficiently.



The answers block (type #9) is an Acnet message block of the form used by the Network Layer software when the answers are to be returned to the requesting node/task. It also includes a pointer to the parent request block (type #13) for use by QMonitor for one-shot requests that need automatic cancellation.



Request message processing also includes building the format block for inclusion in the answer response message. To do this there is a format specification template for each listype included in the LTT module. The template is scanned according to the #bytes of data requested per ident. If the end of the template is reached, and the #bytes requested is not exhausted, the request is in error. This constitutes a new restriction on data requests, where the #bytes that can be requested using a given listype is constrained according to the format spec

After the request support blocks have been filled, the basic request block is inserted into the chain of active data requests using `INSCHAIN`. It is inserted at a position adjacent to another request block made by the same node, if any, in order to increase the likelihood of combining the answer responses of multiple requests into the same network frames. Then the Update Task is triggered to update the request and build the first set of answers immediately.

The request message is processed as it resides in the network frame input buffer DMA'd into memory by the chipset. This processing includes "compiling" the request into the internal ptrs array for later update processing. The message count word in the network frame buffer is decremented to signal to the network that the request message space is now free for future use. Note that initializing the request as it resides in the network buffer (instead of using `NetRecv` to copy it into the caller's buffer) saves copying the ident arrays in the request, at the expense of the additional responsibility of decrementing the message count word when finished with the request message. Of course, both the LCBs and period specification must be copied into the request block for later update processing.

Updating requests

The Update Task scans through all active requests each cycle to update any which are due for processing. It checks for this new request block type (#13) and builds the answers accordingly. The read-type routines are called for each listype using the array of internal pointers to build the answer data. Other data must be placed into the header of the answer message. The format block, however, should remain constant for the request's activity.

A facility for blocking answer responses is specified in the new protocol's period specification. The two parameters given are the maximum number of messages to build before responding and the timeout delay before responding when the maximum number of answers have not yet been built. The size of the answers block is affected by the maximum number of messages parameter, as it lengthens both the format block as well as the answer message itself. As a result, an estimate of the size of the returned answers and the required format block with any blocking is needed before the answers block (type #9) can be allocated.

When the Update Task has built answers that are to be returned to the requester, it invokes the `NetQueue` routine to do it. Just before that, however, it calls `NetXChk` to flush any existing queued messages that are going to a different node or use a different protocol type (different SAP) to the network chipset. This is to ensure prompt delivery of responses to different nodes and yet combine answer messages directed to the same node into the same frame for greater network efficiency.

The Update Task flushes all queued messages to the network after it has processed all active requests each 15 Hz cycle.

D0 Settings

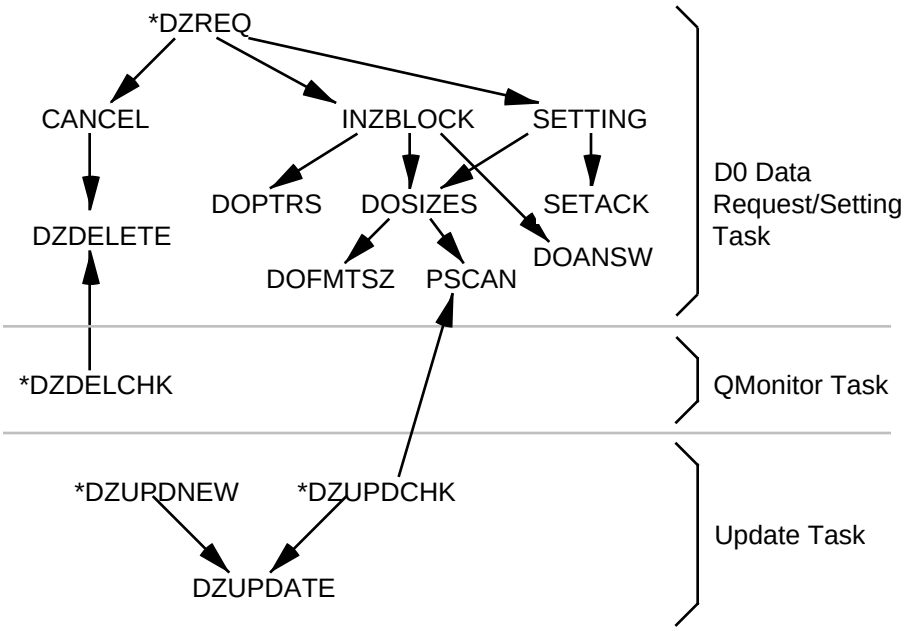
Processing setting messages is greatly simplified because it is all done immediately and because the format of the setting message is nearly identical with that of the request message. The message type word is different to indicate that is is a setting, the period specification is absent, and the setting data offset is specified in the first three words of the message header.

The many set-type routines have been enhanced so that they now return error codes whenever they encounter errors. (Previously, the setting was simply ignored.) This error response word is used in the setting acknowledge message specified in the protocol. A zero value indicates no detected error in performing the setting.

Since setting processing includes an overview scan of the validity of the message, performed by `DOSIZES`, a status-only reply may be given to a setting in place of the setting acknowledge message. For the status-only cases as well as the setting acknowledge cases, refer to the error codes given in the “Error reporting” section of this document.

D0 Request Module Road Map

The organization of the routines in the DZREQ module is as follows, where an asterisk denotes a declared entry point:



The upper collection of routines comprise the DZero Request Task, which waits for a message directed to the destination taskname RPYR and processes it. For a *request* message, the CANCEL routine searches the active list chain for a match against the message id ("list#"), the requesting node and source task id. If it finds a match, it calls DZDELETE to cancel that active request. The INZBLOCK is the bulk of the code which prepares the request block, internal pointers block and answers block for later processing by the Update Task. It uses several other routines to help break that job down into more manageable pieces.

For a *setting* message, the DOSIZES routine is invoked to check for a number of obvious errors. If an error is detected, a status-only reply is given. If not, a doubly-nested loop—outer loop over listypes, inner loop over idents—calls the system routine SETLOCAL to process each setting listype/ident pair. An error return aborts the processing of any remaining settings in the message, and SETACK is invoked to deliver the setting acknowledgment message.

The middle section is the DZDELCHK routine which is called by the QMonitor Task when it has detected the completion of transmission of an Acnet-type message (block type#9) with bit#6 of the NetQFlg word set in the block, indicating that the block is to be retained for re-use. (If the bit were not set, QMonitor would simply free the memory for that block.) It checks for the case of a one-shot DZero data request that should be cancelled. So QMonitor has to

the NetQFlg was set indicating that the block was queued for transmission to the network.

The last section includes two entry points that are called by the Update Task to process type#13 requests during its traversal of the active request chain. DZUPDNEW updates the request only if it has never been updated before, whereas DZUPDCHK examines the period specification and updates the request only if it is time for an update. DZUPDATE shepherds the actual updating of the request and checks the blocking parameters before queuing a response to the network.

Error reporting for requests

A number of potential errors are detected when processing a D0 data request message. For most of these, a response is returned to the requester consisting of a status-only reply, which includes only the Acnet header; neither the format block nor the answer message is attached. Current error codes are as follows:

-64 period spec not implemented yet
-65 invalid message size
-66 invalid request header size
-67 invalid DZero message type
-68 invalid #listypes
-69 dynamic memory unavailable
-70 invalid listype#
-71 invalid identype (error in listype table)
-72 invalid ident length for listype#
-73 invalid #idents for single listype
-74 invalid #bytes requested per ident
-75 invalid offset to ident array in LCB
-76 format block / #bytes conflict
-77 requested #bytes exceeded format spec template
-78 invalid total #idents this request
-79 size of answers format block too large
-80 size of answers too large
-81 #sets of answers too large (blocking spec)
-82 invalid format spec (error in listype table)
-83 request message data offset not implemented
-84 LCB “other” parameters not implemented
-85 spare
-86 setting message data offset out of range
-87 setting message included period spec

In addition to the response to the requester, these errors are recorded in the Local Station in local variables of the DZero Request Task. They can be inspected for

Another error that can be returned by the Network Layer itself is the following:

-21 destination task not connected to network (RPYR not connected)

This means that the 4-byte destination task name in the Acnet header was not recognized by the node that received it. For systems which have Network Layer support but have not yet been updated with the D0 data request software, this will certainly result.

Setting acknowledgment error codes

The following list of errors can occur in response to a data setting message:

0 No error. Setting successful.
1 System table not defined for this listype.
2 Entry# (chan#, bit#, etc) out of range.
3 Odd #bytes of data
4 Bus error
5 #bytes too small
6 #bytes too large
7 Invalid #bytes
8 Set-type out of range (error in listype table)
9 Settings not allowed for this listype
10 Analog control type# out of range (error in analog descriptor)
11 Invalid binary byte address in BADDR table
12 Invalid mpx channel# (Linac D/A hardware)
13 F3 scale factor out of range (motor #steps processing)
14 No CPROQ table or co-proc# out of range
15 Hardware D/A board address odd
16 Bit# index out of range (associated bit control via channel)
17 Bit# out of range for this system's database
18 Digital Control Delay table full (for software-formed pulses)
19 Digital control type# out of range 1–15
20 Co-processor command queue unavailable
21 Co-processor invalid queue header
22 Queue full or unavailable
23 Dynamic memory allocation failed
24 Error status from 1553 controller
25 Invalid 1553 command for one word output
26 Invalid 1553 Command Block address (must be multiple of 16)
27 Invalid 1553 order code in first word of Command Block
28 1553 interrupts not working
29 Cannot initialize 1553 command queue
30 No Q1553 table of pointers to 1553 controller queues
31 Invalid Motor table

35 Invalid data value.
36 Invalid #bytes of text in Comment alarm control
37 No DSTRM table of Data Stream queue pointers
38 Data Stream queue type# out of range
39 Data Stream queue not initialized
40 No MMAPS table of memory-mapped board templates
41 Invalid MMAPS table header
42 Invalid MMAPS table entry size
43 Invalid board# for MMAPS table
44 Invalid directory entry in MMAPS table
45 End of MMAPS table reached during template processing
46 Invalid MMAPS command type code
47 Invalid MMAPS loop params
48 Invalid MMAPS nested loop
49 spare
50 Invalid listype#
51 Invalid ident type# (error in listype table)
52 Invalid ident length for this listype
53 Little console settings switch disabled
54 Little console external settings switch disabled
55 Data Server setting not implemented

Limitations of present implementation

Features *not* supported in the initial version of DZero request handling are the following:

Period specifications *besides* one-shots and simple periodic and blocking
Data offset specified at listype level
"Other" parameters specified at listype level
Error status reporting for each listype-ident pair

It is not intended to support data requests of the "Data Server" type for the D0 protocol. Idents in a request are *ignored* if they do not include the node# of the local station receiving the request in the first word of the ident. This means that one could send the same request to a group of nodes using the functional group multicast form of network addressing, and each node receiving the request would select out its own idents for answer response. (Obviously the requesting node would need to scan the original request in order to be in a position to match the answers with the questions.) Currently, however, the Acnet header-based protocols do not permit sending request messages to a group of nodes.

1553 Data Acquisition

Do it with interrupts

Jul 21, 1988

Introduction

The previous 1553 driver in the VME systems did not use interrupts. It only executed a single command at a time. For data acquisition via an RDATA table entry, the driver waited until the command is finished and distributed the data to the destination data table. This note describes the use of the 1553 board's interrupt capability to allow for simultaneous data collection using multiple 1553 controllers. Note that each 1553 board is in fact a *stereo* controller board, with each of the two controllers able to operate independently with DMA access to its own 64K on-board memory.

There is a separate queue for each controller, and each RDATA entry causes a new entry to be placed into the queue for that controller. If the controller is already active, the new entry waits in the queue until the previous activity completes. If the controller is *not* active at the time the new entry is to be placed in the queue, the command is initiated from the task level to "start the ball rolling." The interrupt routine checks for errors and dequeues the next entry, if any. When no more entries remain in that controller's queue, the interrupt routine exits.

In order that the system does not have to know *a priori* how many 1553 controllers are available, the queueing mechanism is built in a dynamic way. When an entry is to be placed in a controller's queue, a check is first made of a table of queue pointers to determine whether a queue is defined for that controller. The identification of the controller is given by the bits 19–16 of the memory address of the 64K block of memory assigned to that controller. If no queue pointer is found, a new one is entered for that controller, and the queue is initialized empty. The new entry is then placed into the queue.

The data acquisition logic of the Update task must eventually wait for the 1553 controllers' activity to quiet down so the data can be copied from the 1553 memory into the ADATA and BBYTE tables for access by both the Alarm Task and the Update Task logic which fulfills current data requests. A wait-for-completion entry placed in the RDATA table can wait for a single controller or a sequence of controllers' activity to complete. This same entry copies the data from the DMA buffer in the command block(s) into the data tables, taking into account any necessary error processing.

Settings to 1553 hardware also use the queue because the interrupt will occur anyway, given that the hardware switch on the board is enabled, and the interrupt routine must know what to do. For an analog setting, a retry feature could be requested and handled by the interrupt routine's logic. Alternatively,

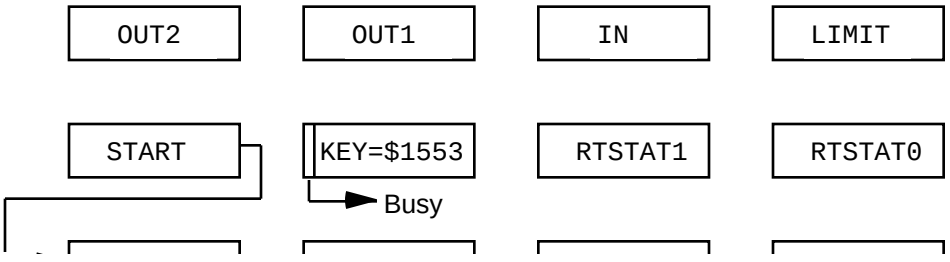
post-processing itself, since these are usually short one-word commands, and there would be no particular advantage to try to overlap them.

Data structures

Data structures are designed for the 1553 queue pointer table and the queues themselves. The queue pointer table is of fixed length—long enough to hold a queue pointer for each 1553 controller in the system. The queue for each controller could be allocated from dynamic memory, but it facilitates diagnostics if it is statically allocated. The queue pointers could be always present for each controller, but that is just one more table to have to get properly initialized. The queue pointer for a given controller could, however, be left in the table to be re-used when the next entry is to be queued for that controller. If the base addresses for the various controllers were dense, the header table could even be indexed by the upper bits of the controller's memory base address. This might make queueing more efficient.

Suppose a queue exists in each controller's 64K memory space. (One must be careful to avoid 32-bit data transfers to the 1553 controller board.) When it is time to place an entry into a queue, the queue pointer is first checked for existence. If it doesn't exist, a queue is created in that controller's memory; hence, one need not initialize the queue ahead of time manually. At this time the interrupt vector# should be written to the `COM1553B` chip and the exception vector initialized in low memory. A base vector# can be assumed by the system—say, $70. The vector# used by a given controller would be this base number plus the controller#. A separate interrupt routine entry point is provided for each controller. The interrupt code can be common after recording the controller# by pushing it on the stack and joining the common code. The queue pointer table is cleared at system reset.

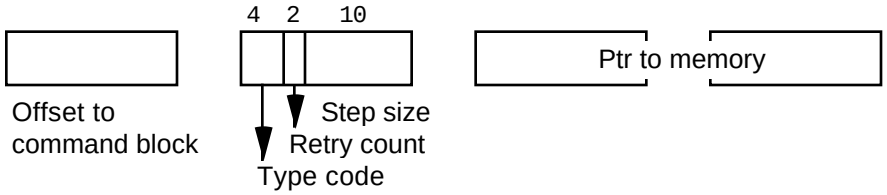
Let's assume that there are no more than sixteen 1553 controllers in one VME system. (To reach this limit would require 8 stereo controller boards of the present design.) The new system table `Q1553` holds an array of pointers to the controller queues used by that system. The table is cleared at reset time. The index into the table is the controller #, the lower 4 bits of the upper word of the controller's base memory address; i.e., bits 19–16 of the 24- bit base address.



an entry in the RDATA table, scans the chain and places only valid command block pointers into the queue for that controller, using the IN queue pointer. If the queue is empty when an entry is placed into the queue, the new command is issued by the CMDS1553 routine itself. This dequeueing uses the OUT1 pointer of the queue. The KEY can verify the existence of a queue.

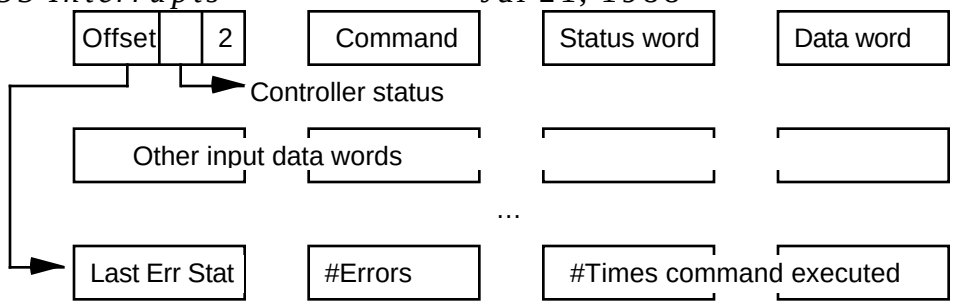
The OUT2 queue pointer user is the code invoked by a later entry in the RDATA table which waits for the queue to be emptied by the OUT1 user—the interrupt routine. It can do any clean-up work required according to a type code in the queue entry, set by the original IN queue pointer user. This could include distributing the 1553 data read in to the various channel or byte entries in ADATA or BBYTE, respectively. It can also copy the setting values into the proper place.

Suppose the queue entry has this format:



The offset word is the low word of the command block memory address, as each controller is allocated 64K of memory space. Since this queue resides in the controller's memory block, the upper word is not needed here. The type code refers to any post-processing that may be required for this entry for use by the OUT2 user of the queue. The retry count is needed for making settings, if retries are to be used. The step size is used in conjunction with the memory pointer to specify the destination memory to which the DMA'd data must be copied to place it into the ADATA table, for example, during post-processing. (Note that this memory pointer must be accessed by words only.) The pointer may also be the data word for a setting. It is better to keep everything dynamic pertaining to the 1553 command in this queue entry rather than in the command block, as the same command block may be queued multiple times to the controller.

The queue can be allocated to reside at any convenient place in the 64K of memory. Suppose we select the end of the memory for this purpose at, say, the last 4K bytes. The offset from the start of the block would therefore be $F000. (Note that the COM1553B chip registers are mapped to reside at the end of the range—from $FFFC–$FFFF.) Using 4K bytes gives room for 500 eight-byte entries.



The layout is shown for a command which inputs a 1553 data word. For the output case, the status and data words are reversed. The first byte is an offset to the diagnostic data kept for each command block. If the Offset=0, there will be no diagnostic block.

Error status

To keep track of 1553 errors, allocate 4 bytes of each controller's memory space in the queue header (`RTSTAT1` and `RTSTAT0`) for storing a longword containing good / bad status for each of the RTs (up to 30) to which the controller can be connected. These may be assigned `BIT` numbers in the VME system, to facilitate reporting of RT-associated errors. (Current 1553 hardware only has errors detectable to the RT level; it does not know about SubAddress-specific errors.) If it is necessary to associate 1553 reading error status with a channel, the relevant RT-related `BIT#` can be stored in the `ADESC` entry. A request for alarm status might include this bit.

Copying readings into the ADATA table:

The 1553 data comes by way of DMA into its own memory; therefore, the data values must be copied into the ADATA table where they are normally found. As a proposal to save this time needed to copy the readings, consider keeping a *pointer* to the reading value in place of the reading value itself. Assume that we allocate 4 bytes for each the reading, setting, nominal and tolerance values, in order to accommodate devices whose readings require more than 16 bits of precision. Each value is a binary two's complement fraction of full scale, as usual. In the case of the reading, the pointer could be stored there instead which points to the data word (or longword) itself. The reading listype could be special code which references the data value indirectly. The problem to solve is how to set this address properly without doing it every time the RDATA entry is processed. If it had to be done each time, no time savings would be realized!

1553 Test Application

A third type of queue entry is used by the 1553 test program. Just as with the settings case, it is unnecessary to overlap test activities; only a single controller is tested at a time. As a diagnostic aid, the first data word value (input or output) can be written to the third word of the queue entry. A single dedicated command block area is used to house each command used by the test program. The command block is prepared, and the queue entry filled. The command is issued, and the routine waits for interrupt activity to complete. As in all the queue entries, the least significant bit of the offset word (to the command block) is set to denote an error, again purely as a diagnostic.

Data Request Timing

Comparison of 3 protocols

Mar 5, 1991

The VME Local Station software supports three data acquisition protocols. The first is the Classic protocol, which was the original protocol developed in 1982. On the token ring network, it uses a special SAP to distinguish it from Acnet header-based protocols. The other two protocols supported are based upon the use of the Acnet header that supports general task-task commun ication across a single network. This note describes the performance by the local station software in its support of all three protocols.

The Classic protocol is based upon *listypes* and *idents*, two abstract specifiers which characterize control system data requests. The design is targeted for distributed systems, in which a single network of local stations connect to the control system signals of a part of an accelerator and contain a local database for their own parts. Because of this, these systems can operate in the absence of a centralized host system. Any host can participate by using the data request protocols.

The second is the DZero protocol, which was developed to satisfy the needs of the D0 control system. It is an evolution of the Classic protocol in that its design is also based upon listypes and idents.

The other is the Accelerator protocol, developed for use with the Fermilab accelerator control system. It was designed to work only with a central ized database. The user program running on a Vax console is given a suite of DPxxx routines that hide the central database accesses used to build a data request.

Two sets of timings were made for each of the three protocols. One was made from the requester's point of view; the other was made from the replier's point of view. The timing was measured by software using a timer of 0.5 msec resolution in the first case and by observing signals available on a front panel connector of the Crate Utility board that show each task's activity in the second.

The example used in the measurements was a data request for readings and setting words of a number of analog channels from one other local station. The number was varied to get the incremental timing on a per channel basis. The test programs are local station console page applications designed for testing each protocol.

The first test measures the time from just before a one-shot request is made until the time that the answers are available in the application's data arrays. For a minimal data request, the timing is about 5 msec independent of the protocol.

The timings for a number of channels > 1 is as follows:

Protocol	#chans	One-shot, msec	Per channel, μ s
Classic	1	5.5	
	510	10.0	90
DZero	1	5.5	
	510	8.5	60
Accel	1	4.5	
	510	23.5	380

A reason for the longer time per channel in the Classic protocol compared to the DZero protocol is that the ReqData routine called by the test application includes a “data server” functionality for local requests, so there is work to be done before the data request message is prepared for the network. In the other two cases, the test program prepares the network message before the timing starts.

The second test measures the additional time spent in the Update Task due to updating the answers in fulfilling the repetitive data request. It does not include any time for transmitting the results across the network. It represents only the additional load on the cpu in producing a set of answers to the data request.

Protocol	#chans	15 Hz, msec	Per channel, μ s
Classic	1	0.4	
	510	0.6	4
DZero	1	0.4	
	510	0.6	4
Accelerator	1	0.4	
	512	0.0	32

The increased time used by the accelerator protocol is because it was not designed for efficient processing. The key concept that is missing is that of specifying an array of idents to be processed under a given listype.

Data Streams

Sucking up message packets

Jan 2, 1989

Introduction

There are instances of the need to access data streams in the VME system. This note explores a generic method of data request for such data streams.

Serial input data

One data stream that has already been supported is the serial input data stream. A request is made for serial data using a specific listype with an ident which gives the serial port that is being accessed. Typically, such a request is made at 15 Hz, specifying a maximum size of the response buffer.

The response data to the request is a data structure consisting of a word containing the number of bytes read from the serial port followed by the bytes themselves. If the number of bytes = 0, then no serial data was collected since the last time. When the number of bytes > 0, the data bytes comprise an integral number of lines of data with each terminated by a carriage return, limited by the size of the requested data. Both nulls and linefeeds are ignored in serial input and do not appear in the buffer.

In the current implementation of serial data requests, only one requester can access a given serial port at once; reading the serial input queue is "destructive." It would be better if more than one requester could sample the serial input at the same time. A data stream protocol can allow for this.

Ramp readback data

Another type of data stream in the VME systems is a ramping co-processor's readback data. In this case, a circular buffer is continuously filled by the co-processor with readings taken at 720 Hz, for example. Requests for this data must be supported. In this case, it may be sufficient to access only the data that is placed into the queue *after* the request is made.

Diagnostic data

One might imagine a diagnostic queue whose contents are to be sampled by a host level diagnostic program. In this case, we need to be able to access data written into the queue *before* the request is made.

Data stream handling

If there were a system table that housed pointers to such data stream queues, then a listype could be designed which addressed one of these streams and sampled what data it found there, returning it at the requested rate.

Response data format

For a general purpose data stream, let us assume that packets are written into

the queue consisting of a size word followed by (size-2) bytes of data. This is exactly the same format used for command packets sent to a co-processor command queue. The response data returned in the requester's buffer would consist of an integral number of such packets. If the size word were zero, no more packets would follow in the response buffer. If there were no data found in the queue, then the first word would be zero, which is entirely consistent with the general format.

Queue types

In order to deal with more than one format of data stream, one can either use different listypes or standardize on a header format which includes a type value. In this way, the processing can vary depending on the queue type. Examples would use different header formats, use pointers rather than offsets, and use both forward and backward "links" to support looking at previous entries in the queue.

Options for data stream access

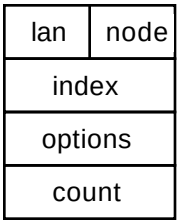
There are several possibilities for copying out the data from the queue to the requester. One case might be to read a raw bytes stream. This could be suitable for a low level access to serial port data, for example. There would be no editing of the byte stream, nor would there be any automatic assembly into lines of text. All such higher level logic would be handled by the requester.

A second example might be access to the raw data, stripping the size word(s) placed in the buffer. The size would have to be assumed known and constant for this to make sense.

And one might like to read old data—that which was placed into the queue before the request was made. To do this, we need to specify how much old data is requested. It could be specified as a number of packets or as a number of bytes. For packets, we need to be using a queue format that includes two "size" words per packet. The first would be the size of the packet, as usual. The second would be the relative offset from the start of the packet (the first size word) to the start of the previous packet. In this way, one can work backwards packet by packet to see previous entries, allowing for the packets to be of different sizes. If the packets were of equal size, and if the offset to the packet located latest in memory were recorded in the queue header, we could look backwards without these size words. But there must be a parameter in the request which specifies either the number of bytes of prior data or the number of packets of prior data.

Ident format

Consider the following ident format:

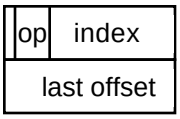


The first word is the lan-node as usual. The second word is the index which selects the desired data stream. The third word selects the options for data collection from the queue. And the fourth word is the count of packets or bytes of prior data requested.

Request processing

In order to allow sampling of the data stream which is non-destructive, one must keep the "last" pointer within the request data block. Currently, there is a single 4-byte pointer used to fulfill a data request associated with each listype-ident pair. To handle data streams, this pointer must keep track of the location of the last entry. But how can it know how to circle back to the start of the buffer? It would seem that 32 bits is not enough.

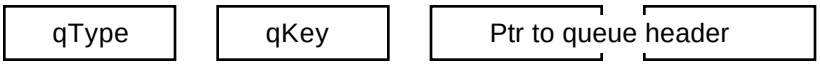
As a solution, suppose the pointer value which is used contains two word-size values. The first is an index into the data stream pointer table. This was given by the ident and would normally be converted into a pointer to the entry in the table. By refraining from converting it to a pointer we save two bytes, which can be used as the second word to contain the offset to the last data packet read from the queue. Now the internal pointer value can lead us to the queue (or data stream) header as well as the position in the data stream where we last left off.



To keep the options value, let the entire second word of the ident be used as the first word of the internal pointer. Then the option value is kept internally to the request. But we can only keep 3 bits for this purpose, as the most significant bit of the first word of the pointer must be used to mark a pointer to an external answer fragment buffer contained within the request.

Data Stream Pointer Table

A system table is used to house pointers to data stream queues. This table is indexed by the index value referred to above.



Data Streams Implementation

Asynchronous packet flow

Sep 5, 1989

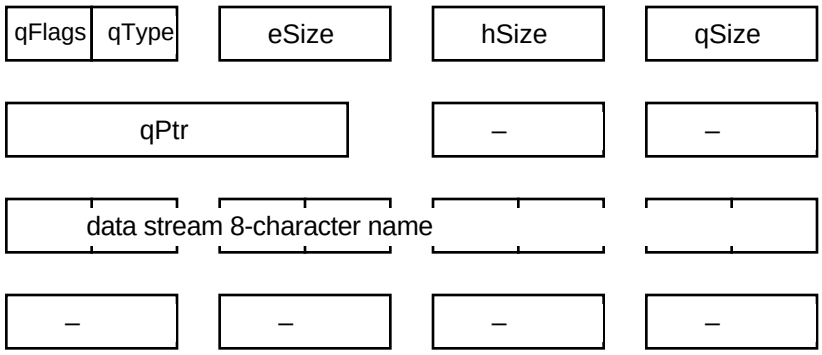
Introduction

Data streams are packets of data that are queued and made available to any data requester. The difference between a data stream and normal data acquisition is that a data stream packet may occur at arbitrary times asynchronous to normal data acquisition. A simple example is data which comes from a serial port. Another is 720 Hz sampled data collected by a ramp co-processor. Another is clock event data.

Normal data acquisition is done synchronously, and typically with only a single value which is collected at 15 Hz. On the other hand, a data stream can have packets added to it at any time even with varying amounts of data. Data stream support herein described makes this variable type of data accessible via a normal data request.

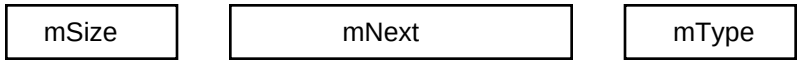
`DSTRM` system table

The `DSTRM` table provides for itemization of the various data streams that are supported. A data stream is identified by an index into this table, just as an analog channel is identified by an index into the `ADATA`/`ADESC` tables. The format of a `DSTRM` entry is as follows:



The `qFlags` include a bit (#6) to indicate that at reset time the queue associated with the data stream should be allocated from dynamic memory. Another flag bit (#7) indicates that the queue has been initialized. The `qType` is a small positive index which gives the type of queue header used, as different types of data streams may require different queue management. This index implicitly characterizes the means of queue initialization, packet entry, and packet extraction. The `Size` word is the entry size of the packets in the queue. For variable size packets, this word is zero. (In this case, the first word of each variable size packet is the size of the packet including the size word.) The `hSize` word is the amount of header space needed to support the data stream itself. It is referred to as the data stream-specific header. The `qSize` is

the total size of the queue which is used to allocate the queue in the dynamic case and is also used to initialize the queue header. The `qPtr` is the pointer to the queue header. In the case of a dynamically allocated queue, this pointer points 8 bytes beyond the allocated area to allow for the common form of dynamic header:

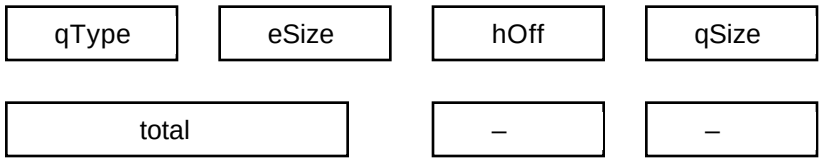


The `mSize` is the allocated size of the memory block, the `mNext` is a pointer to the next block in a chain (when used), and `mType` is the memory block type value of `$000B` for this case. With the `qPtr` pointing just beyond this header, the same `qType` can serve either the dynamic or the static case. This first part of the `DSTRM` entry can be accessed using listype #53.

The 8-character data stream name can be used to identify the data stream mnemonically. It can be accessed using listype #54.

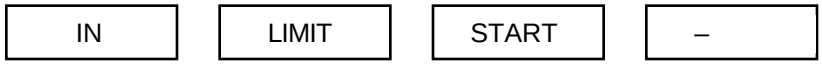
Queue format

The data stream queue format consists of 3 components. The first part is the same for all data stream queues. Its format is as follows:



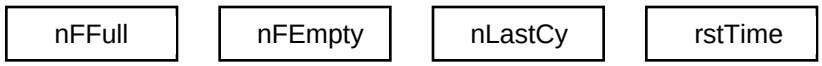
Note that the values are copies of the `DSTRM` entry with a few exceptions. The first word is the `qType` without any flag bits. (This could be changed if the flag bits are needed, as there aren't expected to be many queue types.) The `hOff` is the sum of the header sizes of the first 2 components and is therefore the offset to the data stream-specific header. The `total` longword is the total number of packets ever written into the queue. For diagnostic purposes, the queue header can be accessed using listype #52.

The second component of the queue header is the `qType`-specific header. Its format for `qType`=1 is as follows:



The `IN` word is the offset to the space for the next entry to be placed into the queue. The `LIMIT` word is set to the queue size. The `START` word is the offset to the first entry to be placed. It is initialized to point just after the total queue header.

The third component of the header is specific to the data stream itself. This is the component whose size is declared in the DSTRM entry. An example of the format of the third component is that used for the Clock Event Queue:

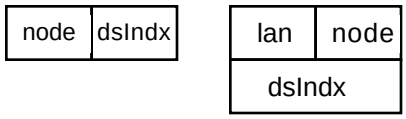


The first 3 words are diagnostic counts which give the number of times the clock event hardware fifo was found to be full (and subsequently cleared), the number of times it was found to be empty, and the number of clock events found in that fifo the last time it was accessed to copy events into the Clock Event Queue. The last word is the time stamp associated with cycle reset that is used to convert the hardware free-running time stamps into ones that are relative to cycle reset. A Data Access Table entry routine manages this header component for the Clock Event Queue.

Additional queue header forms can be designed for other queue types and for other types of data streams.

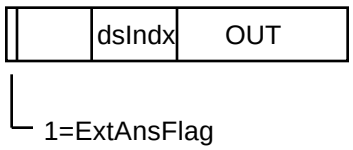
Data requests

A listype (#50) will be used to access data stream packets. The form of ident used is as follows:



Both the short and long ident forms are shown. The requester identifies the data stream index to select the data stream to be accessed. Another listype (#51) is used to request “old” packets—packets which had been placed into the queue prior to the time of the request.

The format of the internal pointer that is kept during request processing is as follows:

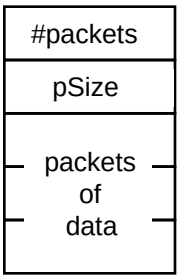


Note that the OUT word, which is the offset into the queue of the last entry extracted is part of the internal pointer and not part of the queue header. This means that different user requests for the same data stream do not interfere with each other. This is a principal feature of the data stream approach. The dsIndx value allows access to the queue header pointer via the DSTRM table for fulfilling the request. When the ExtAnsFlag=1, the rest of the longword is a pointer into an external answer fragment buffer kept with the request,

which just refers to the fact that the data has already been delivered from another node to this node. This last feature is only used for locally initiated requests and for data server requests, not for ordinary network data requests.

Returned data format

The format of the data that is returned in response to a data request of packet data from a data stream queue is as follows:



The first word gives the number of packets that are included in the response data. If it is zero, the queue had nothing in it this time. The second word gives the packet size. If it is zero, the queue uses a variable packet size, and each packet of data will begin with a size word, so the user can process them.

When making a request for previously-written packets using listype #51, the amount of previous packets that can be returned is limited by the size of the requested #bytes. Such requests might be one-shot requests and indicate a large buffer. Requests for only future data might typically be repetitive requests using a moderate size request buffer. A one-shot request to listype #50 would by definition return no information beyond the packet size.

Settings

One can make a data setting to write a packet into a data stream queue. If the queue has variable length entries, a size word (=#dataBytes+2) is inserted ahead of the setting data to form the packet. If the queue has fixed size entries, the length of the setting data must be a multiple of the packet size to be accepted. Either listype #50 or #51 can be used to write a packet into a queue.

The routine `DSWrite` is used to write packet(s) into a data stream queue. It is declared as follows:

```
Procedure DSWrite(dsIndx,nBytes: Integer; VAR data: DType);
```

The `dsIndx` argument is the index part of the ident in the setting request. The `nBytes` word is the number of data bytes, and the `data` parameter is a pointer to the array of data bytes of the packet. If the queue uses variable size packets, only one packet can be written with a single call to `DSWrite`. Note that in this case, a size word is not included as the first word of the data array. The size

word is written (with the value `nBytes+2`) into the queue preceding the packet data.

Settings should not be used to data stream queues other than those which are normally written to by a task. Queues which are written to by interrupt activities or by another processor on the same backplane should not be written to by a data setting.

Software modularization

Most data stream logic is centralized into the `DStream` module. The branch tables indexed by `qType` are all in this module. This includes routines which handle queue initialization, read access and write access. Generation of internal pointers is done as usual by code in the `ReqDGenP` and `PReqDGen` modules.

Data-stream specific code—that used to write into a data stream queue—knows about the `DSTRM` table entry format and the first and third components of the queue header. It does not need to know about the `qType`-specific header component.

Variable size packets

As stated above, variable size packets are recorded in the queue using a size word preceding the data. The size word is sufficient to allow data request processing of the packets using listype #50. But looking backwards to retrieve packets written previous to the request, in order to fulfill a listype #51 request, is quite another matter. In order to make this possible, there is an extra word in the queue that precedes the size word. This word contains the offset from the start of the queue header to the previous packet's size word. This allows backwards traversal of the queue's packets. When a variable packet size queue is initialized, the `START` word points just beyond any data stream-specific header. A zero word is placed there, and the `IN` word points to the next word, which will become the size word of the first packet placed into the queue. The extra previous pointer word that precedes the size word is not returned when packets are delivered in response to a data request.

Data stream-specific header initialization

When a data stream queue is initialized, all data stream-specific header space is set to zero. If nonzero values need to be entered there, the data stream-specific code can notice a cleared value and set up any nonzero initialized values needed.

Digital Control Pulse Delays

What goes up must come down!

8/28/97

Introduction

One might consider that digital control in an accelerator control system consists merely of setting a bit to a "0" or a "1." In hardware terms, this is likely to be what is provided. But when the hardware which is being controlled is taken into account, the situation is a bit more complex.

Turning a power supply on, for example, may be accommodated by a single control line; but often, the supply expects to be driven by pulses rather than a simple level set hi or lo. Whereas a single pulse might drive a reset line to a power supply, it normally takes two control lines to support pulsed on-off control. And, to make it interesting, the on-off *status* of the supply is normally provided as a single status bit.

Timing is important also. The pulses acceptable for control of a large power supply may have to be asserted for a significant fraction of a second, in order for associated relay logic to make up. The required length of the pulse can vary depending on the hardware being controlled.

A facility for generating digital pulses from *memory-mapped* digital output lines has always been part of the VME system software. It was used in the earlier Linac control system for building pulse control of power supply on-off commands, where the length of the pulse might have to be about 0.5 seconds in order to allow for heavy duty relays to make up.

This note concerns extension of the system so that *non-memory-mapped* hardware can also be driven to form pulses—especially for 1553 hardware.

New pulse logic

Since the present memory-mapped pulse-forming logic is part of the 15 Hz interrupt routine, it must be moved to the task level, where 1553 I/O can be handled. A reasonable place to put this logic is in the Update Task, which is concerned with processing entries in the Data Access Table to update the data pool. It could be placed either before or after reading the data. If it is placed *after* reading the data, then there will be fresh readings of the 1553 digital control data to use as a basis for setting or clearing a particular bit.

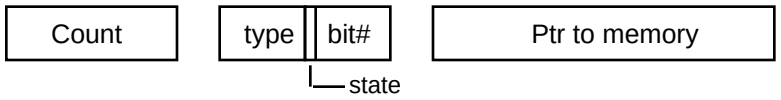
Hardware/software control of pulse delays

The timing of a pulse can be done either in hardware or software. If it is done in hardware, the software must be aware of it, so it will not then try to reset the bit to its non-asserted state after the time delay. And, of course, if it is done in

software, the software must be aware of it and of the required time delay, so it will accomplish the deed of restoring the control line after the delay time has passed.

DCTABLE format

The Digital Control Table is used by the system to support digital pulse delays. It consists of a set of 8-byte entries in the following format:



The count word times out the delay in 15 Hz cycles. The type byte contains the entry type. The entry types used are:

0: *memory-mapped digital control*

In this case, the bit# is a value in the range 0–7, and the Ptr is the address of a byte of memory. When the count reaches zero, the designated bit of memory is either set or cleared according to the value of the state bit.

1: *1553 digital control*

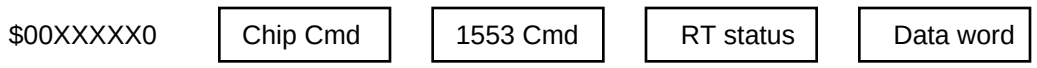
Here, the bit# is a value in the range 0–15, and the Ptr is the address of the data word in the input command block in the 1553 controller's memory. (The input data word is simply the readback of the output data word.) When the count reaches zero, the word of input data is retrieved, the desig nated bit is set or cleared, and the word is output to the 1553 interface.

1553 digital control—background

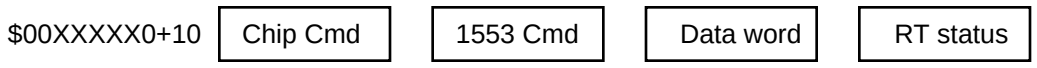
The interface to hardware connected to 1553 remote terminals is handled by the VME system software through command block data structures. These blocks are located in the 1553 controller's own non-volatile memory. Each controller can house up to 64K of memory, and the `COM1553B` chip that actually drives the 1 MHz serial protocol reads and writes this memory via DMA. The command block structure houses the 1553 command word, which contains the addressing information for the RT (Remote Terminal interface to a D0 rack monitor, for example) and provides space for the chip to place a status word and up to 32 words of data it can receive from an RT. For output it contains the data word to be sent to the RT and the status response word.

Control of 1553 digital hardware requires two command blocks as follows:

Input command block:



Output command block:



The command blocks are assumed to always begin on 16-byte boundaries—a software convention. The output command block is assumed to follow the input command block by 16 bytes. So, if the pointer to the input data word is known, the location of the output command block can be determined.

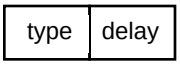
The input data word is sampled, the control bit is set or cleared, and the data word is stored in the output command block and sent to the hardware. The input data word is then updated to match the word which was sent out, in order that subsequent control actions to other bits in the same word can be handled before another reading is made of the input data value. Note that in the case where *hardware* control of a pulse delay is used, this update must *not* be performed, lest subsequent pulse action of another bit in the same word cause a repeat of the first bit's pulse; furthermore, the hardware readback of the control lines must deliver the non-asserted state for any reading of a hardware controlled pulse bit for the same reason.

Software interface for digital control

Two means are provided for delivering settings which result in digital control actions. The first is closer to the hardware; the second is another step away from it.

The first type uses listype #21 to do digital I/O. In this case, the ident used is a Bit#. The 4-byte format is similar to that used for analog channels, but the index value is a Bit# instead of a channel#. The two ident formats are:

The format of the *data* for Bit-style digital control is two bytes in the form:



The digital control type values are:

0: None (no control)
1: Toggle bit
2: Set bit hi (to 1)

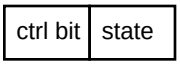
3: Set bit lo (to 0)
4: Pulse bit hi for delay time
5: Pulse bit lo for delay time
6: Pulse one of a pair of bits hi for delay time
7: Pulse one of a pair of bits lo for delay time
8–11: Not used
12–15: Same as 4–7 but hardware-controlled

For this Bit-controlled case, there is no difference between 4 and 6, for example. The Bit# itself must identify the single bit to be controlled.

The delay byte gives the desired pulse delay in 15 Hz cycle units. A value of zero means a "short" pulse. For memory-mapped hardware, this is intended to be about 20 µ sec; for 1553 hardware, it will be much longer than that due to much larger software overhead in the 1553 driver. If the delay has the value of 1, the actual delay time will be less than one 15 Hz cycle, depending upon when in the 15 Hz cycle the command was issued. If a delay of many msec is required, one should probably use at least a value of 2 to insure a delay of at least one 15 Hz cycle. In the case that non-pulsed control is desired, using types 1–3, one may optionally send only the single byte of data for the setting, as the delay byte is not needed.

The second type of command resulting in digital control is the Chan-style digital control. This is done via listype#22, using the channel form of ident. In this case, there is more going on behind the scene. The channel-indexed descriptor entry contains fields which relate both to that channel's associated digital status and digital control. *This is not simple, so bear with me.*

The form of data in the setting command for chan-style digital control is a two byte value as follows:

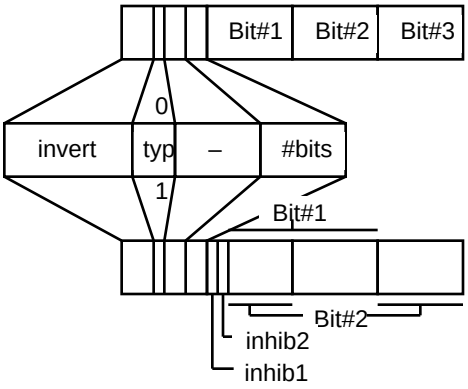


The ctrl bit has a value which designates which of the digital control bits is to be used of those associated with that channel. In the current system, the value of this byte has a valid range of 1–3. The state refers to which of two alternative control actions are to be performed on that bit. Only the least significant bit in the byte is used for this value.

The format of the digital status and digital control fields must both be examined to see what choices there are for these digital control actions.

ADESC digital status field

The format of the digital status field has two variations:



(Note that this is *not* intended to represent the starship Enterprise.)

The 4-byte entry is shown with the various fields tagged for both variations. The first of the four bytes is shown in expanded view for clarity. The invert bits are used by the system software to invert the sense of the raw data that is returned in response to a data request for channel-associated status using listype#5.

There are 3 invert bits which correspond to the possible 3 status bits. These are sampled from most to least significant—that is, left-to-right—in order of successive Bit#'s 1,2 and 3. The least significant pair of bits gives the number of bits related to channel-associated digital status and control for this channel.

Bit#4 of this first byte is used to identify which format type is in use. For format type#0, up to 3 status bits can be supported which are associated with a given channel. The restriction is that they must be Bit#'s in the range 0-255. The parameter page on the small consoles recognizes the value $FF as denoting an inhibit of display of status data for that bit. (It may be used for control only.) This is used for control actions which have no related status bit.

For format type#1, either one or two status bits can be supported. The first Bit# is given by the next two bytes; a second Bit# can be declared using the 2nd and 4th bytes of the 4-byte field. The obvious restriction is that the second Bit# must be in the same 256-bit block of Bit#'s as the first Bit#. In practice, this tends not to be a significant limitation. For this case, the parameter page logic samples the top two bits to get inhibit information about its status display for these bits.

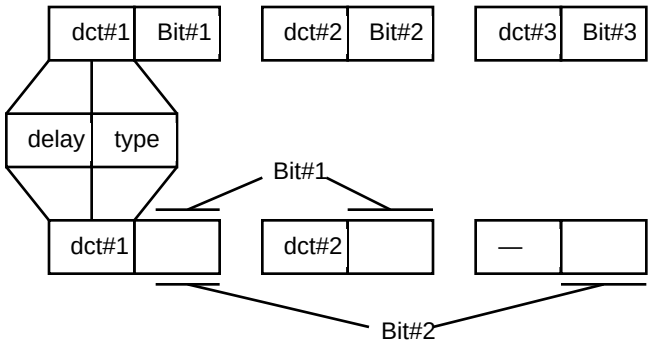
Text for status display

The text used by the parameter page for display of digital status data is found as part of the title text for that channel. When there is a single Bit# used, the last 6 characters of the title (characters 13–18) denote two states—the first 3 characters for the nominal "0" state and the last 3 for the nominal "1" state. (The invert bits are used to reverse this logic from the nominal.) When 2 or 3 bits are used under format type #0, the same 6 characters are used to indicate both states of each bit; therefore, only a single character is available for each state. When 2 bits are used

under format type#1, a second 6-character field is taken from the title for support of display of the second bit's status. This field is characters 6–11 of the 18-character title field. In this case, there is not much room left for any descrip tion of the channel, so the channel's name will have to be sufficient. Of course, all this may change with a revision of the analog descriptor format.

ADESC Digital control field

The digital control field is 6-bytes in length and also exhibits two format types:



As for the digital status case, there is support for up to 3 digital control actions related to a channel, corresponding to the digital status bits described above. If extended Bit# support for Bit#'s > 255 is used, there is support for one or two control actions. The dct values consist of two 4-bit fields. The most significant nibble of the byte is the pulse delay, if needed, and the least significant nibble gives the digital control type# in the range 0–15 described in the table above. All control actions listed in the table can be supported. For types 6 and 7 (and the hardware-driven equivalents 14 and 15, there is reference to a pair of bits used for separate control lines. Such a pair of control lines must be interfaced as a pair of adjacent bits in order that itemizing one bit of the pair implicitly defines the other. Thus, they may be bits 7 and 6, or they may be bits 1 and 0, but they cannot be bits 6 and 5, for example. Either bit of the pair can be entered in the database; the other is found by exclusive or of the Bit# with the constant "1." In the parameter page treatment of this facility of digital control relating to an analog channel, the state value of 0 or 1 is taken from the cursor location at the point of interrupt; if it was in the left half of the status display field, the state value is 0, otherwise 1. It is often referred to as the left/right state for this reason.

Let's have an example

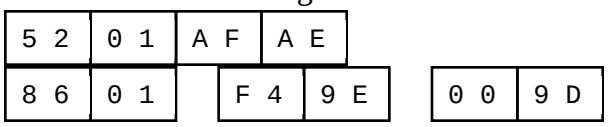
Suppose there is a power supply which supplies current to a magnet and which has an on-off status bit and an "interlocks tripped" status bit, and it uses a pair of on-off control lines plus a "reset" control line which resets the "interlocks tripped" condition. How should the fields in the analog descriptor be programmed to support this device?

Let's not presume that the Bit#'s are small. We must therefore use the format

type#1 described above. Suppose that the Bit#'s are as follows:

Bit#	Meaning
19F	On control pulse (0.5 sec hi-active pulse)
19E	Off control pulse (0.5 sec hi-active pulse)
19D	Reset control pulse (1.0 sec hi-active pulse)
1AF	On-off status (1=on)
1AE	Interlocks (0=tripped)

One possible solution would be the following:



The title text in the local database might be: `"---- OK BAD OFF ON"`
And the parameter page display might appear as follows:
`"MVT102- OK ... ... ON  1013  A  "` *or*
`"MVT102- ...BAD OFF...  0.622 A  "`

Summary

The VME system software supports digital status and control with choices to accommodate a variety of hardware characteristics. The digital data can be memory-mapped bits, or it can be interfaced via 1553. Control bits can be set or cleared or toggled in a "dc" mode, or they can be pulsed hi or lo with a choice upon length of the pulse for software-controlled pulses. Hardware-controlled pulsing is also recognized and provided for. Pairs of bits can be used to provide two control lines that relate to a single status bit.

There is also provision for channel-related status and control bits for the convenience of displays such as the parameter page. Status field text can be used to show the state of status bits, and related control actions can be accepted with only simple setting commands issued from the display program; the details are kept in the local database.

It is hoped that this brief tutorial will serve as an introduction to the variety of digital status and control support provided by the VME systems. In addition, it may inspire host level user programs to devise ever-simpler means of dealing with digital I/O, found in real life to be rather more complicated than simply setting 1's and 0's.

Floating Point Data Requests

Just the engineering units, please!

Mar 3, 1989

The VME systems include a local database of the scale factors for every analog channel. Normally, a data request for the reading of a channel returns the raw channel reading value. The user is expected to separately collect the scale factors and use them to scale the results. This approach has always been taken in consideration of the limited floating point computational abilities of the local stations, which used an 8 MHz 68000 processor. A floating add took about 100 µ s while a floating multiply took 150 µ s.

Now that the stations are being constructed out of 20 MHz 68020 processors plus a 68881 floating point co-processor, it should not require much time to do such conversions to engineering units. The computation required is a conversion of the raw data value to a floating point fraction of full scale, a floating multiply by the fullscale value for that channel, and an optional floating addition of an offset value.

The types of data which need these conversions are readings, settings, and nominal and tolerance values. A new set of four listypes are designed for this purpose. Only systems updated with this feature will be able to handle such requests, as each local station which sources a channel‘s data must do the conversions; it cannot be done only by the requesting node, of course, as the scale factors only reside in the local data base of the original station. Any station will be able to make such requests, however, even if it doesn’t support the feature itself.

In the listype table, the entries for the four new listypes should indicate the table# for the ADATA table, and the ADESC entries must be inferred from it. When these two tables are combined into one, this will be easier. Let the new listypes be 40, 41, 42, and 43, in parallel with 0, 1, 2, and 3. Listype 44 could support the delta setting referred to below.

Systems not using a 68020 could use their software floating point routines for the computations, assuming the use of the feature would not be extensive. Or, such systems could merely decline to support such requests.

Setting support should also be added for these four new listypes. This involves working the linear formulas backwards and watching for the possibility of dividing by a fullscale value of zero. If overflow is reached, the setting should either be clipped to fullscale or not executed at all. A delta setting should also be included for completeness.

The number of bytes associated with the listype used must be 4, or the request is not processed. It could be expanded to allow for 8 bytes also, if double precision values are desired. This would be easy to do, as the 68881 does all the hard work. One would probably have to implement double precision scale factors for this to be meaningful, however.

Note that neither the tolerance conversion nor the delta setting uses the offset value. Reading, nominal, and tolerance use the A/D scale factors, while setting and delta setting use the D/A scale factors. Out-of-range setting values are clipped to the nearest end of range. Or, one might choose to ignore settings which are *way* out of range.

Internal details:

A new read type routine #11 is used to process the engineering units scaling. The new set type routine #13 handles settings with this extra processing, while set type routine #14 handles the delta setting case.

The use of register D5 to hold the field offset in the ADATA entry must be presumed on entry to the setting handler SETENG or SETENGD. This provides the offset needed by the set routine to determine the type of processing required to perform the setting.

This engineering units feature is supported by the three procedures in the (new) READENG module: READENG, SETENG, and SETENGD.

Moderately Fast Data Collection

Up to One Kilohertz with an IRM

Mon, Mar 7, 1994

Introduction

The Internet Rack Monitor (IRM) includes hardware support for 1 KHz digitizing of all 64 analog channels, with possible expansion to 128 channels using two analog interface boards. This note describes a scheme for data request software support of this kind of data, so that it most easily fits into the Classic protocol. The scheme supports a data request for sampling digitized data at rates up to 1 KHz. The data points returned are time-stamped and can provide times relative to a selected Tevatron clock events or times relative to the time of the first data point returned in reply to the request.

Hardware

The hardware writes 64 channels of data each millisecond into a 64K byte circular buffer. This provides room for 512 sets of data, so the circular buffer wraps every half second. A new listype (#82 decimal) is supported for handling access to this data. The ident used with this listype is 3 words in length: the node#, the analog channel#, and an optional clock event specification. From the channel#, the software deduces the area of memory where the 64K buffer resides by adopting a channel assignment convention. We usually assign channels 0100–013F for the first/only 64-channel block in an IRM. In case a second analog board is used, we assign channels 0140–017F to the second 64-channel block. (Actually, any block of 64 channels that starts on a 64-channel boundary is ok, with bit#6 nonzero selecting the second block.) The first block uses memory addressed as 0063xxxx, and the second block uses memory accessed at 0062xxxx. The registers are based at FFF58300 for the first block and FFF58200 for the second block. The following table summarizes this:

Module	*Chan*	*Memory*	*I/O registers*
IP_d	0100–013F	0063xxxx	FFF583xx
IP_c	0140–017F	0062xxxx	FFF582xx

In order to activate the A/D hardware scanning, it is necessary to install a type $28 entry in the Data Access Table. The format of this entry is as follows:

```
2800  InitChan     I/O_register_base
—     —     Mem_base  #Chans
```

As an example,

```
2800  0100 FFF5  8300
0000  0000 0063  0040
```

Only the upper word of the memory base address is given, as its size is 64K bytes. This example serves to copy all 64 readings from the most recent complete set of digitized data in the circular buffer memory into the data pool. The base address of

this memory is 00630000, and the I/O register base address is FFF58300. The presence of this entry in the Data Access Table enables the A/D scanning hardware logic. It also provides 15 Hz sampling in the data pool.

Request protocol

How is the sample period specified? It can be derived from the request reply period and the #bytes of data requested. The requester must provide enough buffer space to hold the data desired, which is central to the Classic protocol, anyway. Given the buffer size and the size of an 8-byte reply header and providing for 4 bytes per data point, the number of points that can be fit in the buffer is known. Given the time between successive replies, and assuming the rate of digitization is 1 KHz, the number of points available in the circular buffer memory is determined. The average time between points is then (#points_available)/(#points_in_buffer). Expressed as a delta set#, including a fractional part, it is used in the loop that maps the data points in the hardware circular buffer into the points to be placed into the user's buffer. Note that the scheme adapts to any variation in the time of updating the data request, because the point last copied is remembered between updates and therefore effectively measures the time since the last reply.

The reply format is then

time_of_first point (4 bytes)
time_between_last_two_events (4 bytes)
array of points as data, time (4*n bytes)

The format of the reply header is two longwords, occupying 8 bytes. The first longword is the time of the first data point in the reply buffer, expressed in units of 10 μ s. The second longword is used only with the clock event# option. If this option is selected, by specifying a nonzero event# in the low byte of the third word of the ident used in the data request, then the second longword is the difference in time between the last two clock events of that kind, also expressed in units of 10 μ s. It is needed to cover the case that the clock event occurred during the time represented by the reply data that follows. The requester should add each point's time value to the time_of_first_point to get the digitizing time of each point. If this time > delta time between the last two clock events, then reduce the time by this delta.

When a periodic request is received, the reply period is used to where in the circular buffer to begin sampling data points. If the period is one cycle, say, then the first reply provides data sampled over the one cycle period just *preceding* the request.

In case the clock event option is not used, the second longword is meaningless to the requester. But the first longword provides the time of the first data point relative to the time of the first data point in the first reply, which is one reply period *prior* to the time the data request is initialized.

The format of a data point is a pair of 16-bit integers. The first word is the data value in the usual two's complement left-adjusted fraction of full scale. The second word is the relative time since the starting time in the reply header, in units of 10 μ s. The relative time value of the first data point in the reply = zero by definition.

Example

Suppose it is desired to collect 100 Hz samples from an analog channel. If the reply period is specified as 1 cycle, and the station runs at 15Hz, then 6–7 points should be collected every 15 Hz cycle. If the reply header is 8 bytes in length, then the requester should ask for 8+4*7=36 bytes of reply data. When it is time to update this request, the front end notices, for example, that 66 data sets have been stored in the hardware circular buffer since the previous update cycle about 1/15 second earlier. To perform the current update, 7 points will be selected out of 66 available. The delta set# is then 66/7=9.42857, on the average correspond ing to about 106 Hz. By accumulating this value in the copying loop and taking the integer part each time to advance to the next set, the 66 points are sampled with 9–10 set spacing. On another update cycle, suppose that 70 data sets have been collected since the previous update cycle. Then the delta set# would be 70/7=10, with no fractional part. This would result in 10 set spacing, or 100 Hz exactly.

If the user specifies a larger buffer in the above case, say 8+4*70=288 bytes, then the calculated delta set# might be 66/70=0.94286, so some duplicate data values would appear in the reply data. In all cases, the reply buffer will be entirely filled. Since the circular buffer wraps every 512 ms, one should not specify a reply period more than 0.5 second (seven 15 Hz cycles or five 10 Hz cycles). Also, the largest buffer size necessary is 8+4*512=2056 bytes, since only the last 512 sets of 64-channel data are stored in the hardware circular memory buffer.

One-shot case

If a one-shot request is issued for such data supported by this listype, there is no reply period indicated. In this case, an attempt is made to reach back as far as possible to collect data to fill the requester's buffer. This might be a good time to supply a large buffer, in order to get the fullest time resolution available for later perusal. This facility allows for a host to monitor for some trip condition, say, and make a request that takes advantage of the hardware circular buffer memory to snapshot the last 0.5 seconds of a signal.

listype, there is no reply period indicated. In this case, an attempt is made to reach back as far as possible to collect data to fill the requester's buffer. This might be a good time to supply a large buffer, in order to get the fullest time resolution available for later perusal. This facility allows for a host to monitor for some trip condition, say, and make a request that takes advantage of the hardware circular buffer memory to snapshot the last 0.5 seconds of a signal.

Memory-mapped I/O for D0

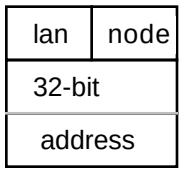
Selective access for DAQ boards

Nov 12, 1991

The online data acquisition used in D0 is based upon VMEbus access to the physics circuit boards. These boards often consist of a complex structure of memory-mapped registers. Some of these registers are normally not addressed at all. Some are read-only. But for downloading purposes, and for reading back to insure that the setting "worked," it is desirable to be able to transfer data to/from the board as if it were accessible contiguously. This note describes a means for the VME Local Stations to handle a variety of specially-configured memory-mapped boards in this pseudo-contiguous fashion.

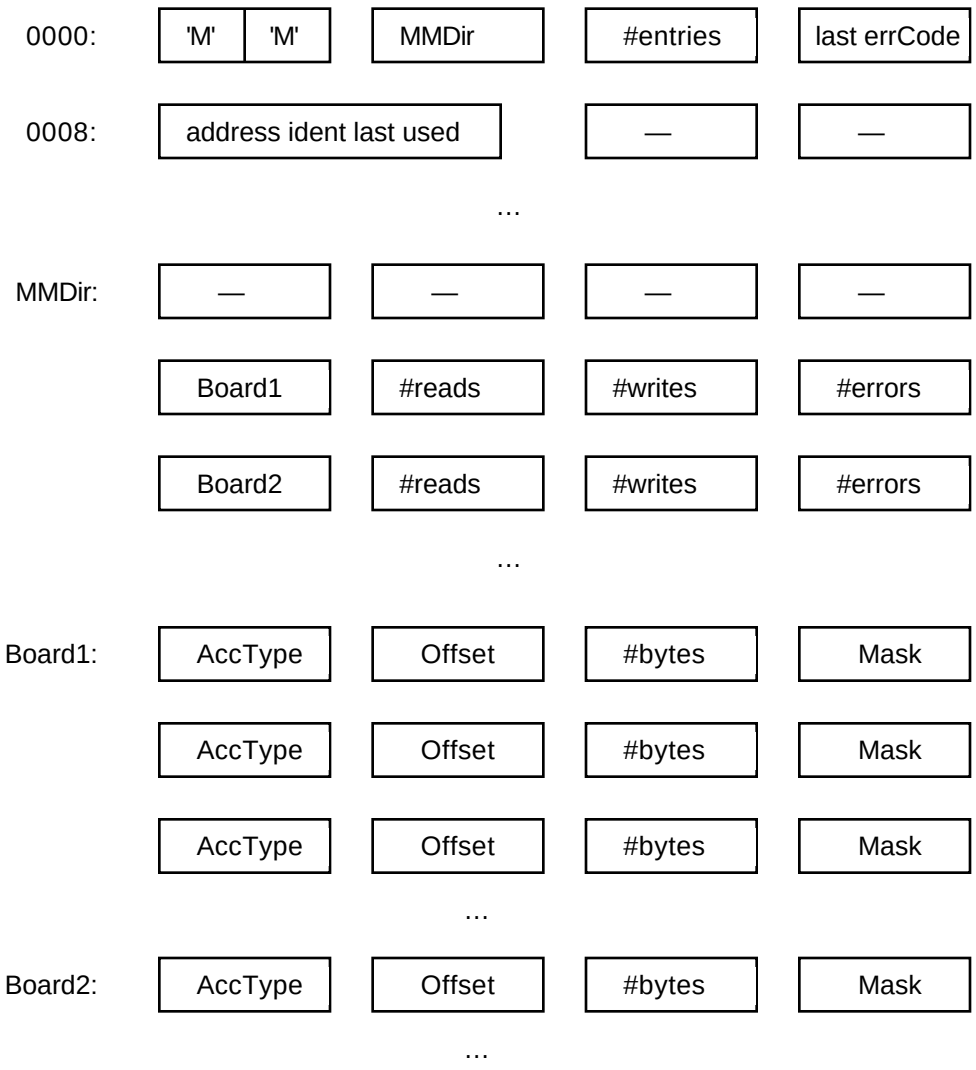
There is a table in the VME systems that holds the memory map information for a number of boards. Listype #46 is used to allow access to boards whose memory maps are described in this table (#16 called `MMAPS` internally). (Listype #47 is used to access the `MMAPS` table itself for downloading purposes.)

The ident used with Listype #46 is a memory address which is the base address of the board. It is expected to have several least-significant zero bits as used on the VMEbus. For this document, let's assume that the board always resides on a 64-byte boundary, at least. This will allow 6 bits that can be used to denote the board type. (It will be seen that this does not limit the accessibility beginning at areas of the board which are offset from the base address.) This allows a single listype to be used to access several different boards—up to 63. And it means that the D0 database can contain the address portion of the ident in 4 bytes. The ident format is as follows:



Again, the low order few bits of the address hold the board type number. The actual base address of the board is assumed to end in just as many zero bits.

MMAPS Table Memory Layout



In the `MMAPS` table format, the first word is a key to check for a valid table. The second word specifies the offset to the first directory entry indexed by board type#. (Board0 is not used in order to check for the presence of the board type in the low order bits of the ident address.) The third word gives the number of board types which are supported. The fourth word is the last nonzero error code produced. (A list of these error codes can be found at the end of this document.) The next longword is the last address ident used in read or write access using the `MMAPS` table.

From the board type, the directory entry is found consisting of an offset (from the start of the table) to the array of command entries that describe the memory map for that board plus some diagnostic counters, including counts of successful read and write accesses and a count of errors. The number of commands to describe each board can be variable; the directory entry only tells where the first command begins.

Each command may consist of two basic forms, each consisting of four words of information. The normal form includes an access type code to describe how the memory is accessed. The only access type initially supported is type=1, denoting a sequence of memory words. The second word is an offset value to be added to the "current" target address before processing this command. The third word gives the number of data bytes to be processed for that command. The fourth word allows for a mask that identifies the read / write bits in each word processed for purposes of verifying a successful setting. At the con clu sion of the processing of a command, the current address points to the next location beyond the block, allowing for the loop processing described next.

The second form of command is the loop command. The first word of a loop command is `$D0D0`. (Read this as "D-zero DO" loop—small joke.) The second word gives the number of subsequent commands that comprise the body of the loop. The third word specifies the loop count. Loops can be nested.

As an example of a series of commands to access a hypothetical board which has 16 groups of 64-byte blocks, in which it is desired to access only the 3rd–5th and the 25th–31st words. The loop command form would be used for the 16 groups, and the two sets of registers would be specified by one command for each. A third command would serve to skip to the end of each group. The following specification could be used:

D 0 D 0	0 0 0 3	0 0 1 0	—
0 0 0 1	0 0 0 4	0 0 0 6	F F F F
0 0 0 1	0 0 2 6	0 0 0 E	F F F F
0 0 0 1	0 0 0 2	0 0 0 0	F F F F

Note that, as a crude check, the sum of the non-loop command offsets and #bytes words should add up to the length of the block in bytes. For this example, one would specify that the number of bytes needed to access the board was (6+14)*16= 320.

As each command is processed, a check is made that the #bytes remaining to be processed as specified by the user is large enough to cover the block described by the command. If it is not, then the number of bytes of data to be processed in the block is reduced to match the number of bytes remaining. All this means is that the bytes requested can end anywhere within the command processing. In addition, if a bus error is encountered during access to any word of memory, further processing is terminated at the listype level. This can happen if the board is not plugged into the crate, for example, or if the vertical

interconnect is unplugged.

The scheme described here makes it possible to accommodate the memory maps of several D0 physics boards. Note that different memory layouts can be handled for the same physics board. The VME system doesn't really know about boards; it only knows about memory maps given by entries in the MMAPS table. The table contents can be entered manually, or they can be downloaded by a host utility program. The format of the table is designed for efficient processing by the VME system software. The inner loop for processing the command which specifies a block of memory words is as small as it is for normal memory access.

Diagnostic error codes in MMAPS table header or a board's directory entry:

2:Invalid MMAPS table entry size (odd or < 8 bytes)
3:Invalid board# (> #entries)
4:Invalid board directory entry (offset to command list)
5:End of MMAPS table reached during command processing
6:Invalid command type code
7:Invalid loop command parameters or invalid nested loop
8:Verify failure after entire setting
$4000:Bus error occurred during access to target memory

Error codes returned to CDAQ are 42–48 for the above codes. In addition are these relevant error codes:

40:MMAPS table doesn't exist in VME station
41:MMAPS table is invalid (key ≠ 'MM')
4:Bus error occurred during read or write access
7:Invalid #bytes
52:Invalid ident length

Message Queue Formats

How does the message queue work?

Sep 14, 1988

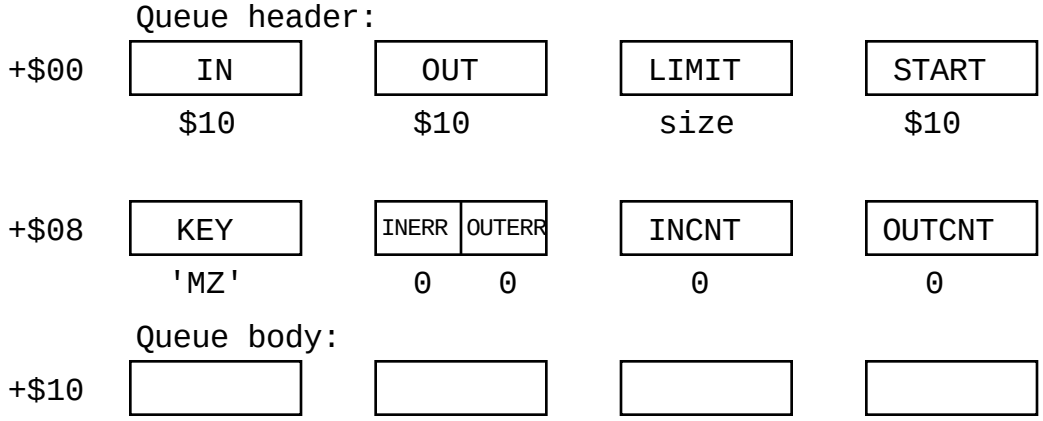
This note describes the format of the shared memory Message Queue used for communication with another processor on the VMEbus. It is implemented as a simple one-way queue. Messages are placed in the queue by the VME System computer. The other processor removes messages from the queue and interprets the command accordingly.

Initialization

The VME System computer initializes co-processor message queues at reset time. One of the standard system tables—table #15—contains pointers to the message queue for each co-processor. This queue pointer table, indexed by co-processor number, has 8-byte entries of the following format:

The message queue pointer is followed by the total queue size.

Each co-processor queue has the following format:



The values beneath the words in the queue header are the initialized values. The other processor, when it recognizes the presence of the queue, examines the KEY field. If it has the value 'MZ' it changes it to 'MQ' to signal to the VME system cpu that it has "seen" the message queue. (Until this happens, the VME system cpu will not place messages into the queue.)

Queue header

The IN pointer (offset from the start of the queue header) points to the next available space in the queue for a message. It is altered by the VME system cpu as the last act upon placing the new message into the queue, after first

checking that there is room available to hold the message.

The OUT pointer points to the next message to be removed from the queue by the co-processor. It is altered after the co-processor has removed the message from the queue. When the two pointer IN and OUT are equal, the queue is considered empty. When they are unequal, there is at least one message in the queue.

The LIMIT word is the total size of the queue (in bytes). It is determined by the contents of the VME system table directory.

The START word is the offset to the start of the queue body. When new entries have reached LIMIT, the IN pointer circles back to START.

The KEY word has the value 'MZ' when the queue is initialized, and it is changed by the co-processor to 'MQ' to signal that the queue has been recognized.

The INERR byte counts times when the VME system cpu tried to place an entry into the queue, but found the queue full.

The OUTERR byte is incremented by the co-processor cpu when it encounters an error in processing the messages it removes from the queue.

The INCNT word is incremented for each message successfully placed into the queue.

The OUTCNT word is incremented by the co-processor when it successfully removes a message from the queue.

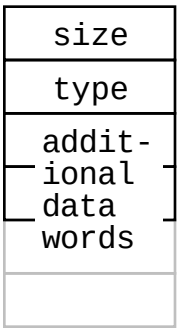
Protocol

When the VME system cpu has a message to place into the queue, it checks to see that the KEY word has the value 'MQ'. It then checks to see if the message can fit. If IN≥ OUT, it checks for space between IN and LIMIT; if there is not enough space there, a zero is placed at the word pointed to by IN, and IN is reset to START. If either IN<OUT or IN had to be reset to START, it checks for space between IN and OUT. If there is room, the message is copied into the space, and the IN pointer is advanced by the message size.

The co-processor examines the queue at its convenience. (Note that the co-processor had to have *a priori* knowledge of the location of the queue.) If the queue is not empty (IN≠ OUT), a message is removed from the queue. The word pointed to by OUT is examined. If it is zero, OUT is reset to START, and if IN≠ OUT, the word at START is examined. When the co-processor has

removed the message from the queue, it advances the `OUT` pointer by the message size. It may then check to see if another message is present. It is assumed that the co-processor will poll the message queue often enough that the queue will not fill up. If it does, one will find the `INERR` count nonzero.

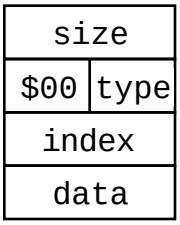
Message format
Messages placed in the queue for a co-processor conform to a simple structure:



The first word is the message size, including the `size` word. The second word is the message `type`. Additional message contents may follow the second word. So the minimum message size is 4 bytes, in the case that no additional data is required for a given type. The `type` word and any additional data have meaning only to the co-processor, not the VME system.

Generic message setting
To send a general message to a co-processor in a VME system, one has only to send the appropriate setting, specifying listype #40. The ident used with this listype supplies the co-processor number (0,1,2…). The number of data bytes specified in the setting—incremented by 2—becomes the `size` word of the message placed into the selected queue. The first word of the setting data, then, is the `type` word of the message. Additional data words follow the `type` word. Note that the VME system does not care about the `type` word value.

Analog control
There is a format in use for analog control, in which an analog channel may be set which results in a message sent to a co-processor queue.



In this case, the `type` value is given by a byte from the analog control field of the analog descriptor for that channel. The `index` value is given by a word

from the analog control field. The data word is the word of setting data in the analog control setting. The `size` value is therefore 8 bytes.

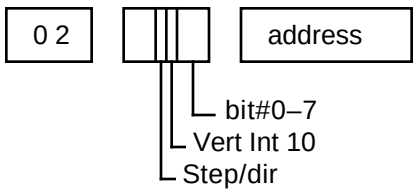
Motor Control Specification

Analog control field parameters

Jul 14, 1990

Motor control is specified via the contents of the analog control field of a given channel's analog descriptor. A number of cases are supported to handle various forms of motor I/O interfaces. This note describes the features available and how to specify them.

The form of the analog control field for a motor is as follows:



The analog control type byte is $02 for motors. The values of the second byte provide for most of the variations. If the value of the byte is ≥ $20, then the last 3 bytes are interpreted as a pointer to a 1553 command block in the range `$200000` to `$EFFFF0` which can be used to send the two's complement count of motor steps to the hardware. In this case, the hardware is assumed to generate the pulses to step the motor automatically.

If the upper 3 bits of the second byte are zero, then we have the case of a memory-mapped digital I/O motor interface, and the local station cpu will arrange to deliver the motor steps pulses of about 20 µ sec duration to the motor at 150 Hz. The rest of this note deals with variations of this memory-mapped case.

The least significant two bytes of the byte address used for motor control are given by the last two bytes of the field. The upper two bytes of the byte address are assumed to be `$FFFF` to indicate VME short I/O space in the local station crate. A variation of this is specified by bit#3 marked "Vert Int 10" above. If this bit is set, the upper two bytes are assumed to be `$10FF`, to indicate access to VME short I/O in a slave crate accessed via the Vertical Interconnect hardware. The slave crate is assumed connected to VI card#1, port#0. This feature allows testing Vertical Interconnect access.

Bit#4 is used to distinguish two types of hardware control of memory-mapped motor interfaces. When it is zero, the bit# indicates the CW pulse bit, and the adjacent bit (bit# exclusive-OR'ed with 1) indicates the CCW bit. For example, if the bit# given is 7, then the CW pulse is connected to bit#7 and

the CCW pulse is run by bit#6.

When bit#4 of the second byte is a one, the bit# indicates the step pulse bit, and the adjacent bit is the direction control bit. Simultaneously, the bit# refers to the CW limit switch status, and the adjacent bit is the CCW limit switch status, where 1=limit switch active. This limit switch status byte is located either 1 or 2 bytes from the control byte, depending upon whether the address given is even (1) or odd (2). The Ironics digital I/O board is interfaced to odd bytes only, whereas the BurrBrown board is interfaced to consecutive bytes. To make this work with the latter board, motor control bytes should be connected to even byte addresses only.

To specify whether pulses should be low active or high active, the least significant bit of the bit# is used. If the given bit# is odd, high active pulses are issued. If it is even, then low active pulses are used. Since the bit also indicates which one is used for CW pulses (or for the step pulse), one cannot switch polarities independent of the hardware connections. So this factor must be considered when planning the wiring to the motor interface.

Summary

Several variations of motor control can be specified via proper settings of the analog control field. Motors can be controlled via 1553 interface, Ironics or BurrBrown digital I/O boards, CW/CCW or step/direction interfaces, hi-active or lo-active stepping pulses, and optional access via the Vertical Interconnect hardware to a slave VME crate. Providing more support variations than these may require a larger analog control field or indirect access via another table of specification parameters.

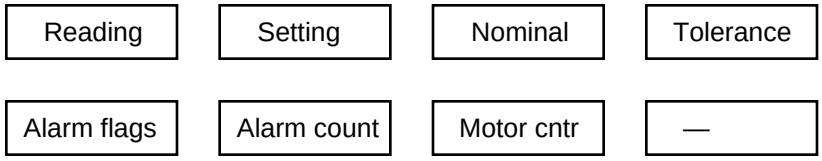
System Tables

An introduction to their uses

Sep 27, 1990

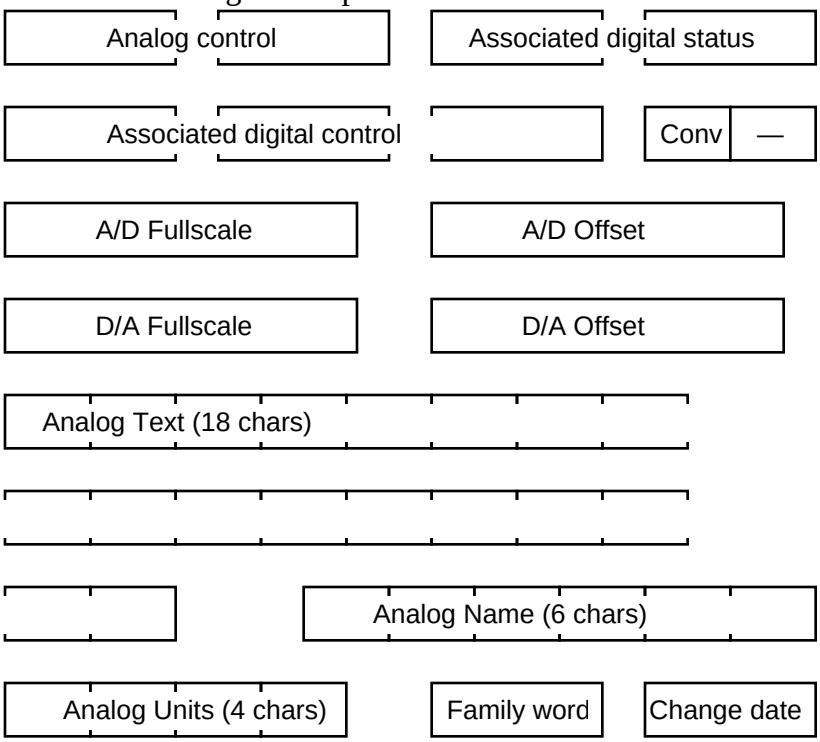
The VME Local Station software makes heavy use of system tables, which are statically allocated arrays of records mostly in non-volatile memory. Each table is specified in the system table directory, whose entries are indexed by system table#. This note introduces each system table and describes its use.

#	NAME	Description	Indexed by	#bytes/entry
0.	ADATA	Analog Data	CHAN	16



Analog channel dynamic values are kept here. The latest analog reading, the last setting successfully issued and the analog alarm scan parameters comprise this entry. The motor countdown word is also used for capture data. (See Data Access Table entry type $16.) This table is the analog data pool. See the document "Alarms Task" for more details about Alarm flags.

1.	ADESC	Analog Descriptor	CHAN	64

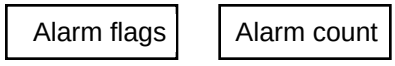


The analog local static database includes these fields for each channel. Additional fields used for D0 alarms are in AADIB. The date-of-last-change for this entry is encoded into 16 bits as year(7)/month(4)/day(5).

See the document "Analog Control Types" for details on the analog control field. See the document "Digital Control Pulse Delays" for more details on

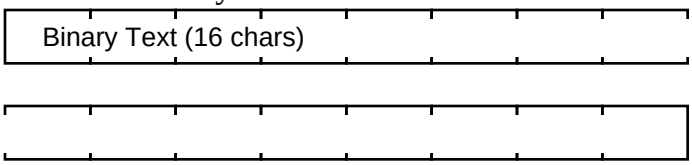
Associated status and control bits. See the document “Related Groups of Channels” for more about the Family word. See the document “VME Station Database” for more about the local database in general. (The latter document is fairly old, however.)

2. BALRM Binary Alarm BIT 4



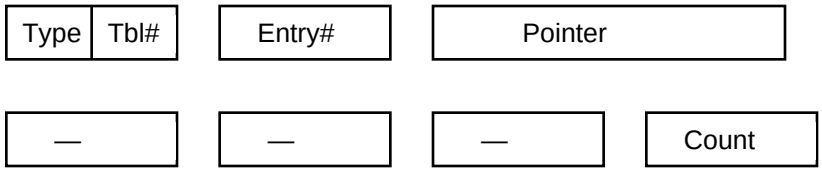
The alarm flags for each binary Bit are the same as those for an analog channel, except that bit#14 of the word is used for the nominal bit state.

3. BDESC Binary Bit Titles BIT 16



This is a 16-char text description of each binary Bit.

4. RDATA Read Data Access Table Indx 16

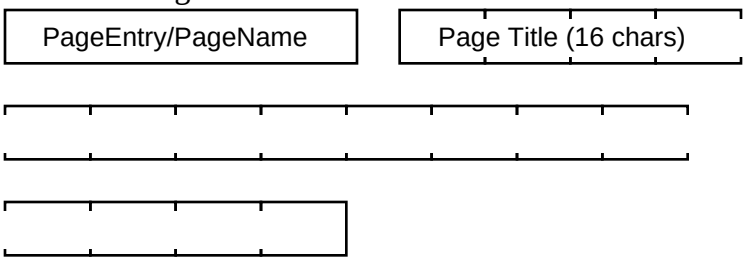


This entry is difficult to illustrate since there are so many formats used depending upon the Type byte. See “RDATA Formats” and “RDATA Periodicity” documents for many more details.

5. BBYTE Binary Status Bytes BYTE 1

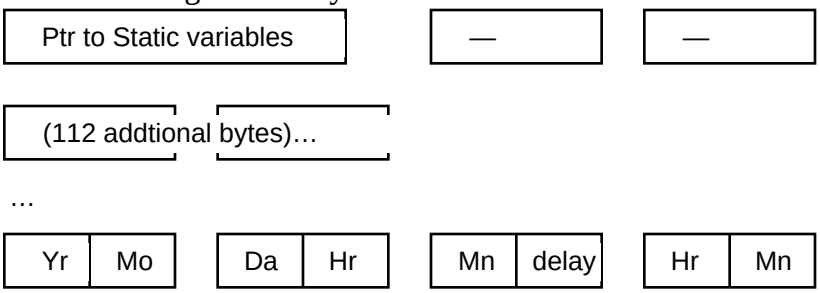
Each entry is a single byte that is read (via the $04 Data Access Table entry) based upon the contents of the corresponding entry in the BADDR table. This table constitutes the binary data pool.

6. PAGEP Page Pointer PAGE 20



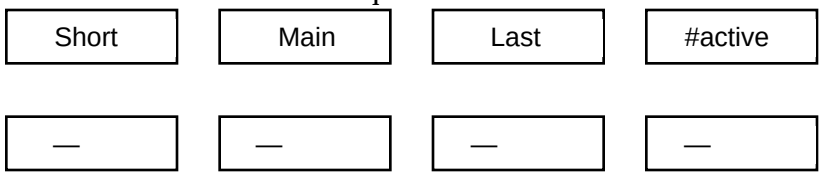
The first 4 bytes may be either a pointer to the entry point for the page application or the 4-char name which, when prepended with PAGE, forms the named program key to the CODES table entry that refers the page application program in non-volatile download and volatile executable memory. The 16-character title displayed for this page is also included here. See the document “Local Station Applications” for a list of page applications and see separate user guide documents about each one.

7. PAGEM Page Memory PAGE 128

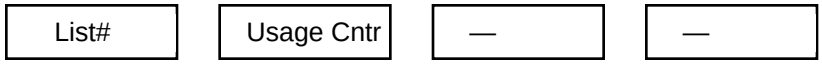


Each page application is given 120 bytes of page-private memory that is saved across display page invocations. The example shows the first 4 bytes being used to house a pointer to a static variables record that is used for variables whose values must be saved across successive cycles while the display page is active. The last 8 bytes of the 128-byte entry are used for the auto-page parameters, which allow automatic and periodic invocation of a display page. The first 5 bytes hold the Next time the page will be called up, while the last 2 bytes specify a Delta periodic time, which may be 0000 for a one-shot auto-invocation. The delay byte specifies a timeout (in the range one 15 Hz cycle to 4 minutes) after which the system will automatically terminate the automatic invocation of the page. See the document “Auto-page on Demand” for more details on the auto-page facility.

8. LISTP Active Data Request Ptr Indx 8



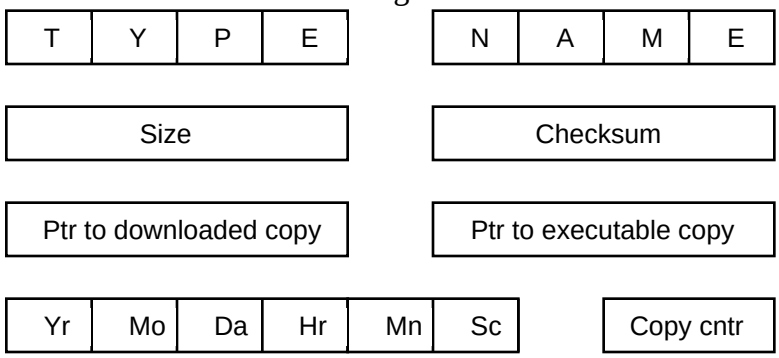
The header of this table includes an offset to the Short and Main sets of entries as well as the offset to the Last vacant entry used in the Main set. The number of active requests includes locally-generated and data server requests, but it does not include ordinary network data requests.



The Short set portion of the table includes the “real” 11-bit List# associated with the byte size list# specified by the ReqData caller followed by a usage count of this associated byte size list#.

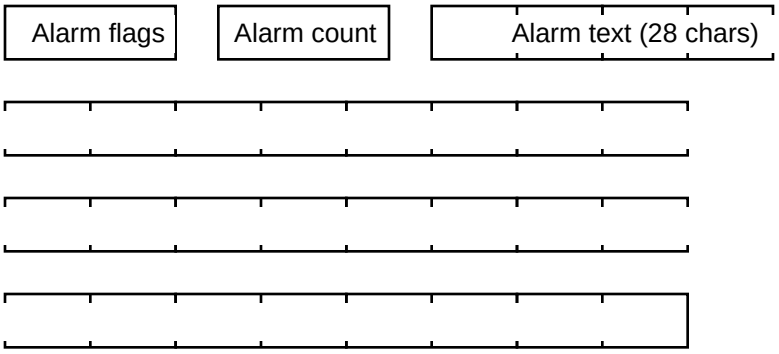
The Main set portion of the table includes a usage counter of this table entry and a ptr to its active request support memory block. The latter is zero if the entry is inactive. This table is for internal system use only.

9. CODES Downloaded Programs Indx 32



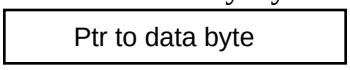
Each entry denotes a named downloaded program. The first 4 chars are used to indicate different types of programs; *e.g.,* PAGE is used for page applications and LOOP is used for local applications. Within a type, a 4-char name mnemonically denotes a particular program code. Each valid entry contains a ptr to the downloaded copy in non-volatile memory and an optional ptr to volatile executable memory used while the program is active. The download time is recorded along with a diagnostic count of the number of times that space is allocated in the executable area for a copy of the downloaded program.

10. CDATA Comment Alarm Data CMNT 32



This table includes the static text that is used for Comment alarms along with the usual alarm flags and count to permit bypassing such messages. As of this writing, the only Comment alarm is a system reset message; this table makes provisions for more such alarm messages.

11. BADDR Binary Byte Address BYTE 4

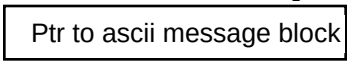


The Data Access Table entry type $04 causes all entries of this table to be scanned and each data byte read placed into the BBYTE table. If the hi byte of the ptr has the value $80, the lo 24 bits are considered a ptr into a one-word 1553 command block data byte.

12. OUTPQ Output Pointer Queue Indx 8

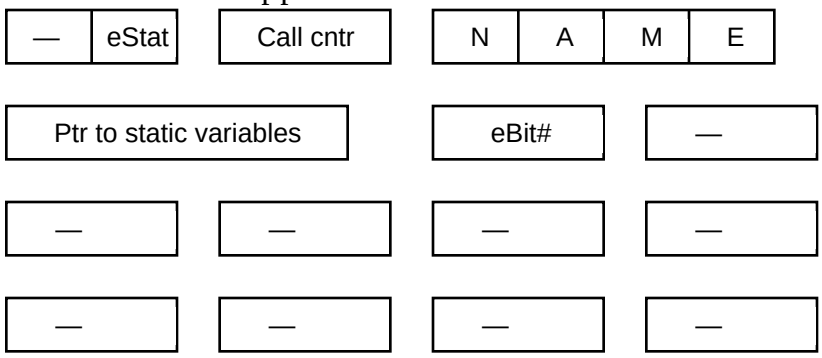
This table includes an entry for every message that is destined for the network. The sign bit of the first byte is a “used” bit that is set when the entry has been passed on to the network. The other 7 bits are a delay counter used for a timeout on network response.

13. PRNTQ Serial Output Queue Indx 4



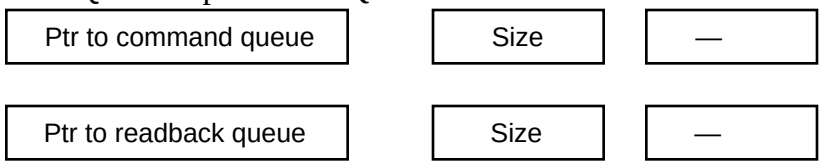
The message block includes one line of ascii text to which a CR-LF is appended. This queue allows spooling of lines of text for serial output.

14. LATBL Local Applications Indx 32



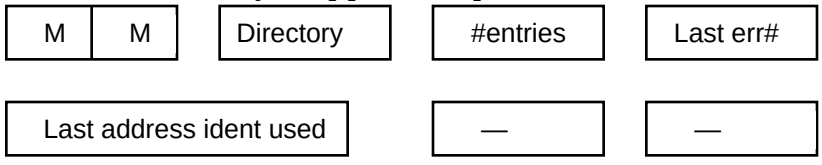
Local applications are the means of supporting closed loops in the local station. Each entry characterizes an instance of a closed loop, allowing for multiple uses of the same closed loop code with different parameters. The code to be invoked is known by 4-char name. When it is called, a ptr to the last 24 bytes of this entry is passed. This provides a means of storing a ptr for the static variables used by this instance for use in subsequent calls while the loop is active. The eBit# is the enable / disable Bit# used for control of the closed loop instance. The eStat is the memory of its last value to detect changes in the enable / disable state. The Call cntr is a diagnostic count of the calls made to the code.

15. CPROQ Co-processor Queue Ptrs COP 16

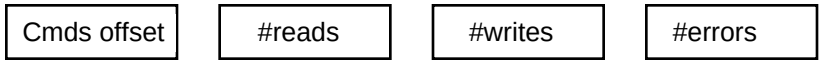


A co-processor is a cpu that shares the same backplane with the local station. A simple one-way message queue is used for communication with the CoP. Provision is made in the entry for a read-back queue, but as of this writing, no system support yet exists for such.

16. MMAPS Memory-mapped Template Indx 8



This table supports gather-read and scatter-write access to memory according to commands in a template. Multiple memory layouts can be supported by separate entries. The header of the table includes a key to recognize a valid MMAPS table, an offset to the start of the directory portion of the table, The #entries in the directory and some error diagnostics.

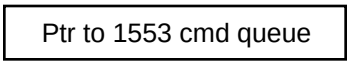


For each directory entry, the offset to the list of commands that describe the detailed layout of the memory is given, along with some statistics.

For each command, the type of command entry is followed by the offset to the data to access, the #bytes to access and the #bytes to skip. For the loop type command entry, the following layout applies:

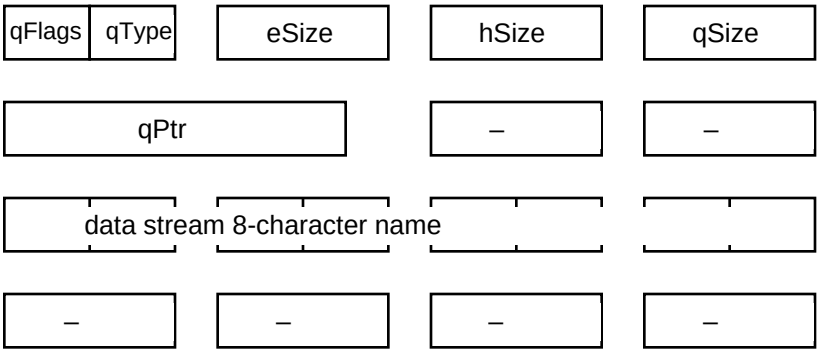
The #cmnds is the number of commands following to be looped over. See the document “MMAPS Table Entries for D0 Boards” for more details.

17. `Q1553` 1553 Controller Queue Ptrs `C1553` 4



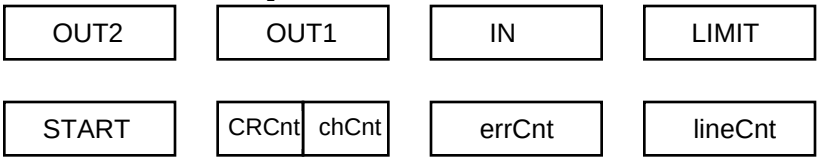
For each 1553 controller is use, a command queue is initialized and a ptr to it placed into this table. The queue itself is in the memory on the 1553 controller board, currently at offset $F000 from the start of a 64K block. (If the controller has only 16K memory, it is effectively at offset $3000 from the start.) See the document “1553 Data Acquisition” for more details about this command queue. Also see the document “VME 1553 Interface” for more about 1553 command blocks and related topics.

18. `DSTRM` Data Streams `STRM` 32



There is an entry in this table for each data stream in the system, indexed by data stream number. At reset time, the data stream queues are initialized according to valid `DSTRM` entries. Only qType=1 is currently supported. The eSize word allows for fixed or variable size queue packets. The hSize word is the size of the *data stream-specific* header component of the queue header. The qSize includes the space for the queue header. The qPtr points to the data stream queue header. See “Data Streams Implementation” for more details.

19. `SERIQ` Serial Input Queue — 1K

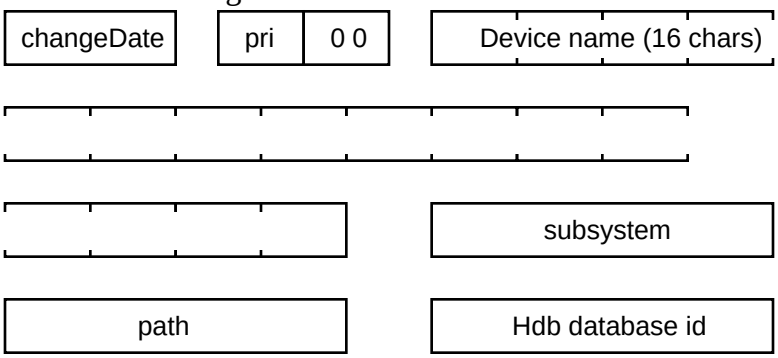


The header for this queue includes offsets for the first and second “output users” and for the “input user” of the queue. The `LIMIT` word is the size of the queue, and the `START` offset is where the first character will be placed. Characters received via serial port interrupt are placed into this queue, advancing the `IN` offset. Nulls and LFs are ignored. When a CR has been read

and placed into the queue, an event is sent to activate the Serial Task. It advances the OUT1 offset of the queue. A user requesting listype #36 data advances the OUT2 offset, this consuming the data read. (A data stream implementation of serial port access could eliminate this side effect.) See the document “VME System Serial Port Handling” for more details.

20. ----- spare

21. AADIB Analog Alarm Device Info Blocks CHAN 32



This table supports auxiliary information required by D0 alarm messages for each analog channel The last 30 bytes of each table entry is downloaded from the Host’s database. A date-of-last-change is automatically recorded in the first word in the 16-bit format of Year(7)Month(4)Day(5). See the document “D0 Alarms” for more details.

22. BADIB Binary Alarm Device Info Blocks BIT 32

(See AADIB for layout.) This table supports auxiliary information required by D0 alarm messages for each binary bit.

23. CADIB Comment Alarm Device Info Blocks CMNT 32

(See AADIB for layout.) This table supports auxiliary information required by D0 alarm messages for each comment alarm index.

24. ----- spare

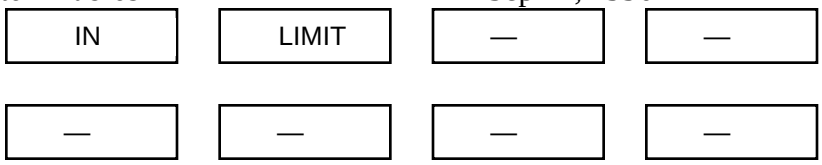
25. ----- spare

26. ----- spare

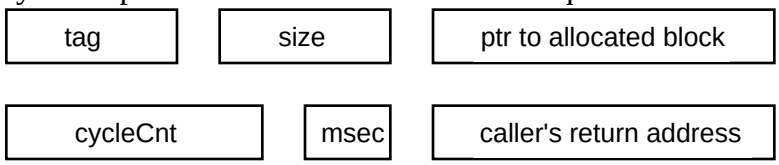
27. ----- spare

28. ----- spare

29. DIAGQ Alloc, Liber Diagnostics Indx 16



The header of this *optional* diagnostic queue includes the offset to the next entry to be placed and the total size of the queue.



Each entry denotes the occurrence of a call to Alloc or to Liber to allocate or free a block of dynamic memory. The tag word has values of `$AA00` for an Alloc call and `$FF00` for a Liber call. The size of the block allocated or freed and the ptr to the block follow. The time of the call is specified as a count of 15 Hz cycles since reset and a count of 0.5 msec since the start of the current cycle. The last longword is a copy of the return address of the call to Alloc or Liber.

30. `TRING` Token Ring Network — 8K

All token ring network functions are organized around the use of this table, the structure of which is fairly complicated. Its layout is presented in some detail in the document “VME Token Ring Table.”

VME System Data Requests

Mar 15, 1989

Introduction

One of the most important services provided by the VME system software is support for handling requests for data. Each VME station is able to respond to many active data requests simultaneously. Each request may ask for data to be returned repetitively at rates specified in sub-multiples of 15Hz. One-shot requests, in which only a single response is given, are also supported. This paper describes how requests for data are handled by a station on the network.

VME Station Data

Data which is collected by a station is kept in tables in memory. Most data requests call for copies of selected data from these tables. One of the most popular tables is the Analog Data Table. An entry in this table contains the most recent reading, setting, nominal and tolerance values for each channel known to that station. The entries are indexed by channel number. (A channel herein refers to a single analog quantity known to the system in its local database, another table of entries also indexed by channel number.)

Values of 16-bit readings are collected from the hardware at 15Hz, according to the instructions contained in the Data Access Table. The last setting value is recorded in the Analog Data Table upon successful setting of the associated control hardware, such as a D/A. The nominal and tolerance values are specified by the user and are used by the alarm scanning logic. That code checks new data readings against the nominal value within the specified tolerance to judge whether a channel is in a "good" state or a "bad" state. Each time a change of state occurs, an alarm message is sent to the network node which is designated to process them.

A common type of data request, then, is to request readings of a selected group of channels. The support for data requests is optimized to respond to repetitive requests for a given type of data from a random selection of table entry identifiers. The type of data of data is specified in a data request by a "listype" number. This is a small integer currently in the range 0-39. The table entry identifier is called an "ident." It consists of a network node number followed by an entry number. So, the listype indicates the type of data requested, and the ident specifies which table entry to access.

In order to take advantage of the way the system is optimized to respond to requests, one should specify a listype and an array of idents for which that type of data is being requested. When updating the response to a request, the internal VME software works with an array of pointers, one pointer for each ident in the request. The array of pointers is processed according to code optimized to handle the particular type of data which is accessed using the specified listype.

The listype mechanism is actually somewhat more general than stated above. A listype number indexes into an internal table which contains a reference to code to generate the internal pointers which are used during response update, a reference to code which processes a setting (if allowed) for that type of data, and parameters which

specify the table and entry offset appropriate for that listype. But it is possible for data to be accessed which does not come from simple table entries. For example, data may be requested from a serial port. In that case, the response data consists of a word containing the number of characters followed by that number of characters, limited to a maximum length as specified in the request. (In this case, an integral number of lines of text is returned, where a line ends in a carriage return code.) Another example is a request for binary status data. One may request such data by using a listype which expects a list of bit number idents. In this case, the response consists of a single byte with the value of 0 or 1 for each bit number, according to the current state of the status bits as found in the Binary Data Table (the digital "analog" of the Analog Data Table), which contains the latest readings of the binary status data bytes.

Sources of Data Requests

VME stations can receive requests for data from several sources. Each station can support a local console application which may make requests by calling system procedures. The simplest case would request a list of data using only idents of the local node. The second case occurs when a local request includes idents which indicate a node number different from that of the local station. The local station software then issues the request to the network. If the array of idents includes idents from more than one "other" node, then the request is broadcast in order to reduce network traffic, at the expense of requiring all stations, not just those identified in the request, to examine the request. So, a network node may receive a request for data from another station who made the request. In this case, the station(s) examine the received request for idents which match their own station number, and they gear themselves up to respond only to their own part of the request. The requesting VME station, then, has the job of combining these "response fragments" into a single response for the requesting application program.

A third source for data requests is via the Data Server. A network node may make a Data Server request to any VME station. Such a request is treated internally just like a locally-generated request. It may consist of idents of any set of stations. The Data Server Task will collect the response fragments that are returned to its node and deliver them to the original requesting node. It is recommended for network efficiency reasons that the station chosen to be the serving station be one of the stations which is mentioned in the request, perhaps the node specified in the first ident mentioned in the request.

A fourth source for data requests is via the Serial Server. In this case, the request is sent from a requesting computer to the serial input port of a VME station. The Serial Server then makes the request locally. When the response fragments are collected and ordered, a serial response is transmitted back out the serial port to the requesting computer.

Acnet Data Requests / Settings

System Implementation

Mar 16, 1990

Introduction

The message formats for Acnet data requests / settings is described in the Acnet Design Note 22.28. It uses the Acnet header designed by Charlie Briegel to support generalized task-task communications across a network. The Network Layer software in the VME Local Stations supports these Acnet header-based messages. This note describes the implementation of the support for Acnet Data Services data acquisition and setting messages.

Message flow

When a request or setting message is received, it is directed to a well-known taskname RETDAT for requests and SETDAT for settings. (These 6-character network task names are encoded in the "Radix-50" form used by PDP-11 computers.) At initialization, the Acnet Request Task creates a message queue (called ACRQ) that is used to receive Acnet header-based messages directed to the taskname RETDAT or to the taskname SETDAT. NetCnct registers both tasknames to the Network Layer. (By directing both message types to the same queue, processing of the messages in original network order is assured. One can issue a setting command and immediately issue a request to read back the setting value and still be confident of obtaining the new setting, assuming a valid setting.)

```
Function NetCnct (taskName, queueId, eventMask, VAR taskId);
```

The eventMask is left zero, as the Request Task will simply wait on the message queue rather than wait on an event. The Request Task then enters an infinite loop that calls NetCheck to wait for a message and, upon receiving one, processes it.

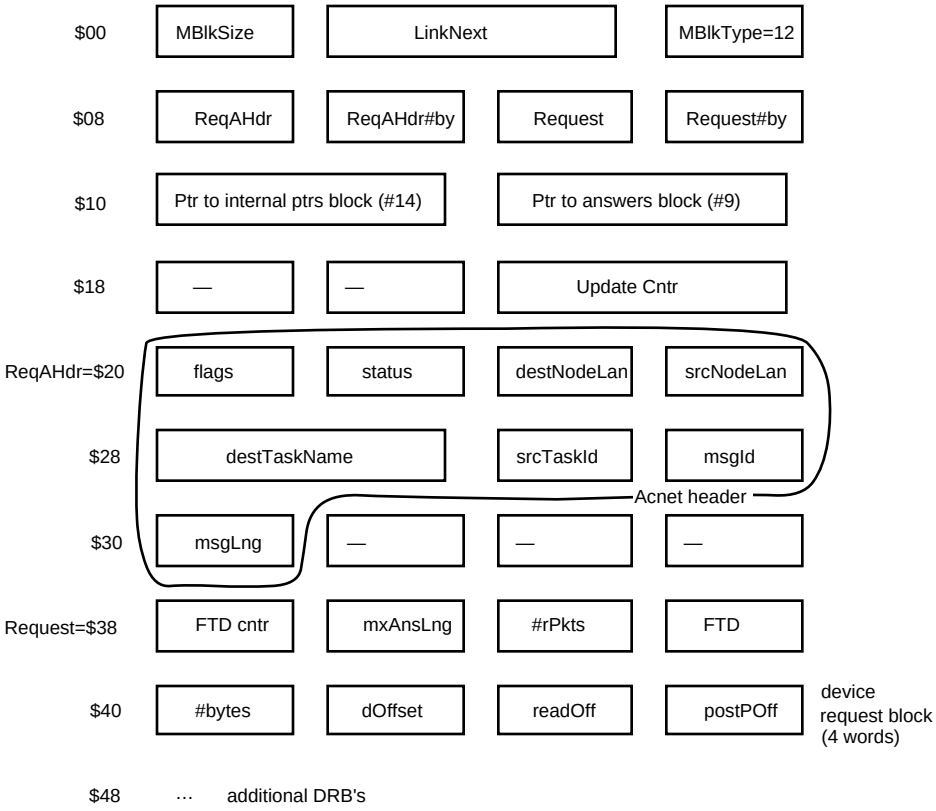
```
Function NetCheck (taskId, timeOut, VAR msgRef);
```

When the function returns with valid status, the message type is checked as found in the first word of the Acnet header. If it is a USM (unsolicited message) with the CAN bit set that was directed to RETDAT, the request identified by the message id is cancelled. If it is a USM that was directed to SETDAT, the setting message is processed immediately with no acknowledgment message.

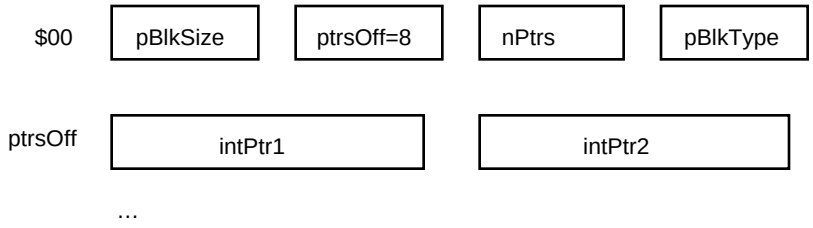
If the message type is a request, the message following the header is checked. If it is a setting, it is processed immediately, and an acknowledgment is returned in the form of a status-only reply message (Acnet header only). If it is a request for data, then 3 message blocks are allocated for support of the new request. (If the request specifies an existing active message id, then the existing request is cancelled.) The basic request block (type#12) houses the various parameters

needed to monitor the request activity. Two pointers are included in that block that point to the other related allocated blocks—the internal ptrs block (type#14) and the answers block (type#9).

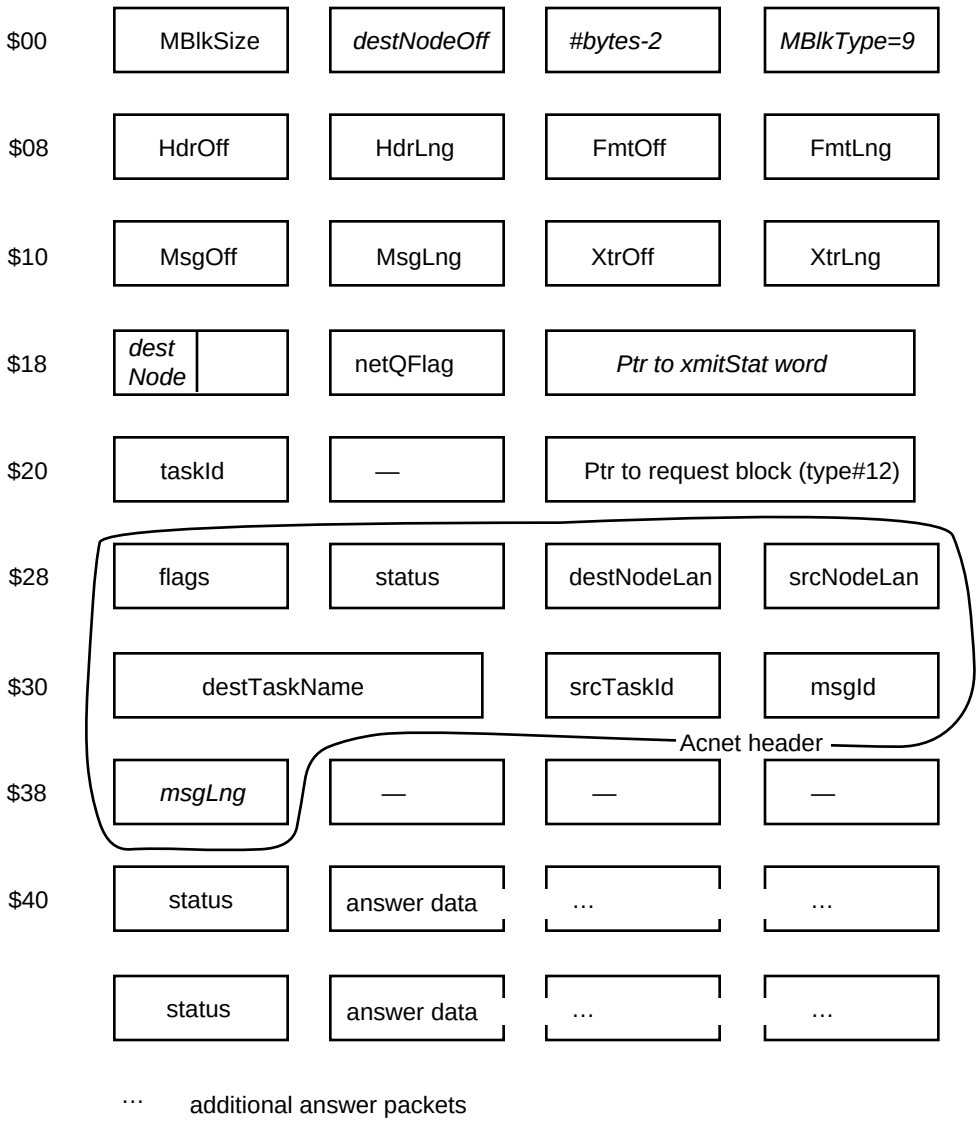
The basic Acnet request block (type #12) contains the array of device request blocks (DRB's) and the frequency time descriptor (FTD).



The Internal Ptrs block (type #14) contains the array of internal ptrs that are used to update the request (build the answers) efficiently.



The answers block (type #9) is an Acnet message block of the form used by the Network Layer software when the answers are to be returned to the requesting node/task. It also includes a pointer to the parent request block (type #12) for use by QMonitor for one-shot requests that need automatic cancellation.



After the request support blocks have been filled, the basic request block is inserted into the chain of active data requests using `INSCHAIN`. It is inserted at a position adjacent to another request block made by the same node, if any, in order to increase the likelihood of combining the answer responses of multiple requests into the same network frames. Then the Update Task is triggered to update the request and build the first set of answers immediately.

The request message is processed as it resides in the network frame input buffer DMA'd into memory by the chipset. This processing includes "compiling" the request into the DRB's and the internal ptrs array for later update processing. The message count word in the network frame buffer is decremented to signal to

the network that the request message space is now free for future use. Note that initializing the request as it resides in the network buffer (instead of using NetRecv to copy it into the caller's buffer) saves copying the ident arrays in the request, at the expense of the additional responsibility of decrementing the message count word when finished with the request message.

Updating requests

The Update Task scans through all active requests each cycle to update any which are due for processing. It checks for this new request block type (#12) and builds the answers accordingly. The read-type routines are called for each listype using the array of internal pointers to build the answer data.

When the Update Task has built answers that are to be returned to the requester, it invokes the NetQueue routine to do it. Just before that, however, it calls NetXChk to flush any existing queued messages that are going to a different node or use a different protocol type (different SAP) to the network chipset. This is to ensure prompt delivery of responses to different nodes and yet combine answer messages directed to the same node into the same frame for greater network efficiency.

```
Function NetXChk (newNode, newType): Integer;

Function NetQueue (taskId, VAR msgBlk, VAR xmitStat): Integer;
```

The Update Task flushes all queued messages to the network after it has processed all active requests each 15 Hz cycle.

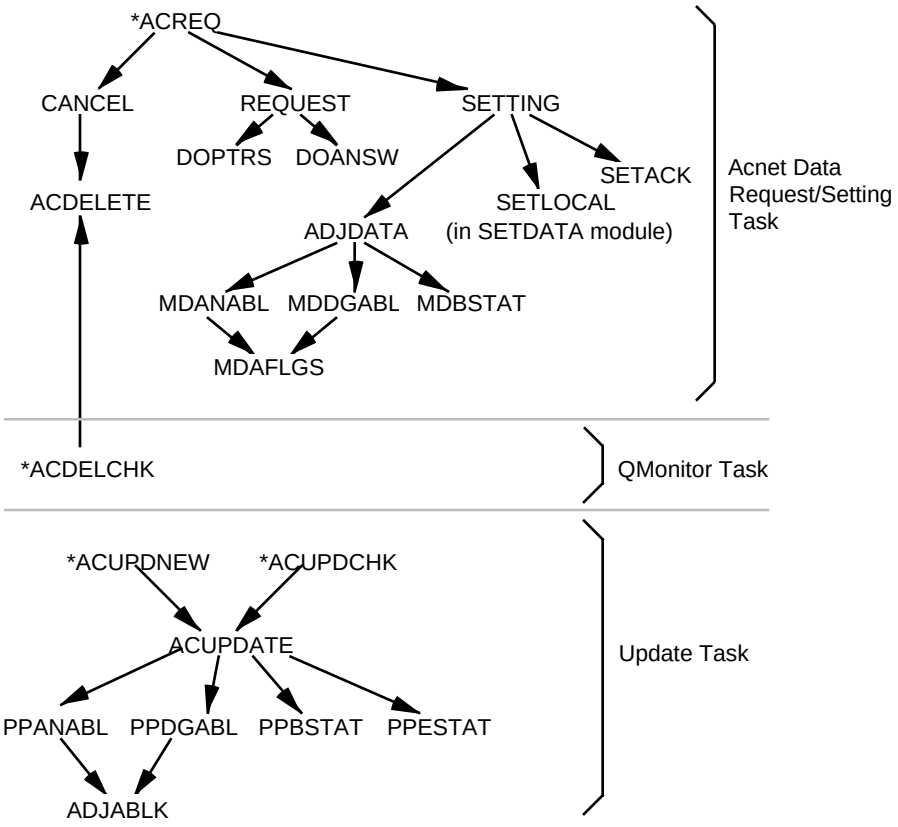
Acnet Settings

Processing setting messages, as compared with data requests, is greatly simplified because it is all done immediately and no dynamic data structures need be prepared for later update processing. The destination task name of SETDAT indicates that the message is a setting.

The many set-type routines have been enhanced so that they now return error codes whenever they encounter errors. (Previously, the setting was simply ignored.) This error response word is used in the setting acknowledgment message. A zero status indicates no detected error in performing the setting. This acknowledgment is returned only if the setting message type is a request. If the setting message is a USM, no acknowledgment is returned.

Acnet Request Module Road Map

The organization of the routines in the ACREQ module is as follows, where an asterisk denotes a declared entry point:



The upper collection of routines comprise the Acnet Data Request Task, which waits for a message directed to the destination taskname RETDAT or SETDAT and processes it. For a *request* message, the CANCEL routine searches the active list chain for a match against the message id ("list#"), the requesting node and source task id. If it finds a match, it calls ACDELETE to cancel that active request. The REQUEST is the bulk of the code which prepares the request block, internal pointers block and answers block for later processing by the Update Task. It uses several other routines to help break that job down into more manageable pieces.

For a *setting* message, the setting action is performed immediately. The system routine SETLOCAL is called to process each packet. An error return aborts the processing of any remaining settings in the message, and SETACK is invoked to deliver the setting acknowledgment status-only reply message.

The middle section is the ACDELCHK routine which is called by the QMonitor Task when it has detected the completion of transmission of an Acnet-type message (block type#9) with bit#6 and bit#5 of the NetQFlg word set in the block, indicating that the block is to be retained for re-use and that it is a *Acnet* protocol request as opposed to a *DZero* protocol request. It checks for the case of a one-shot Acnet data request that should be cancelled. So QMonitor has to recognize the type#9 message and be aware of the NetQFlg word. It also looks

for the case of the type#$F9 and frees the memory of that block. (A type#$F9 block is an altered type#9 block no longer needed for holding an Acnet answer response but could not be freed when cancelling the Acnet request because bit#7 of the `NetQFlg` was set indicating that the block was queued for transmission to the network.)

The last section includes two entry points that are called by the Update Task to process type#12 requests during its traversal of the chain of active requests. `ACUPDNEW` updates the request only if it has never been updated before, whereas `ACUPDCHK` examines the FTD counter and updates the request only if it is due. `ACUPDATE` shepherds the actual updating of the request and queues an answer response to the network.

Error reporting for requests

A number of potential errors are detected when processing an Acnet data request message. For most of these, a response is returned to the requester consisting of a status-only reply, which includes only the Acnet header. Current error codes are as follows:

-32 spare
-33 invalid message size
-34 spare
-35 invalid #request packets
-36 dynamic memory unavailable
-37 invalid listype#
-38 invalid identype (error in listype table)
-39 invalid ident length for listype#
-40 invalid #bytes requested per ident
-41 invalid total #idents this request
-42 size of answers too large
-43 size of answers > max length given
-44 nonzero data offset not supported in request packet
-45 nonzero data offset not supported in setting packet
-46 invalid #setting packets
-47 invalid read routine type# (error in listype table)
-48 node# does not match this system's node#
-49 invalid destination task name

In addition to the response to the requester, these errors are recorded in the Local Station in local variables of the Acnet Request Task. They can be inspected for diagnostic value (with suitable instruction). For each error, a data word is recorded for the last error of that type followed by a count word of the number of errors of that type that have occurred since the station was reset.

Another error that can be returned by the Network Layer itself is the following:

-21 destination task not connected to network (RETDAT or SETDAT)

This means that the 4-byte destination task name in the Acnet header was not recognized by the node that received it. For systems which have Network Layer support but have not yet been updated with the Acnet data request software, this will certainly result.

Setting acknowledgment error codes

The following list of errors can occur in response to a data setting message:

0 No error. Setting successful.
-65 System table not defined for this listype.
-66 Entry# (chan#, bit#, etc) out of range.
-67 Odd #bytes of data
-68 Bus error
-69 #bytes too small
-70 #bytes too large
-71 Invalid #bytes
-72 Set-type out of range (error in listype table)
-73 Settings not allowed for this listype
-74 Analog control type# out of range (error in analog descriptor)
-75 Invalid binary byte address in BADDR table
-76 Invalid mpx channel# (Linac D/A hardware)
-77 F3 scale factor out of range (motor #steps processing)
-78 No CPROQ table or co-proc# out of range
-79 Hardware D/A board address odd
-80 Bit# index out of range (associated bit control via channel)
-81 Bit# out of range for this system's database
-82 Digital Control Delay table full (for software-formed pulses)
-83 Digital control type# out of range 1–15
-84 Co-processor command queue unavailable
-85 Co-processor invalid queue header
-86 Queue full or unavailable
-87 Dynamic memory allocation failed
-88 Error status from 1553 controller
-89 Invalid 1553 command for one word output
-90 Invalid 1553 Command Block address (must be multiple of 16)
-91 Invalid 1553 order code in first word of Command Block
-92 1553 interrupts not working
-93 Cannot initialize 1553 command queue
-94 No Q1553 table of pointers to 1553 controller queues
-95 Invalid Motor table
-96 Motor table full

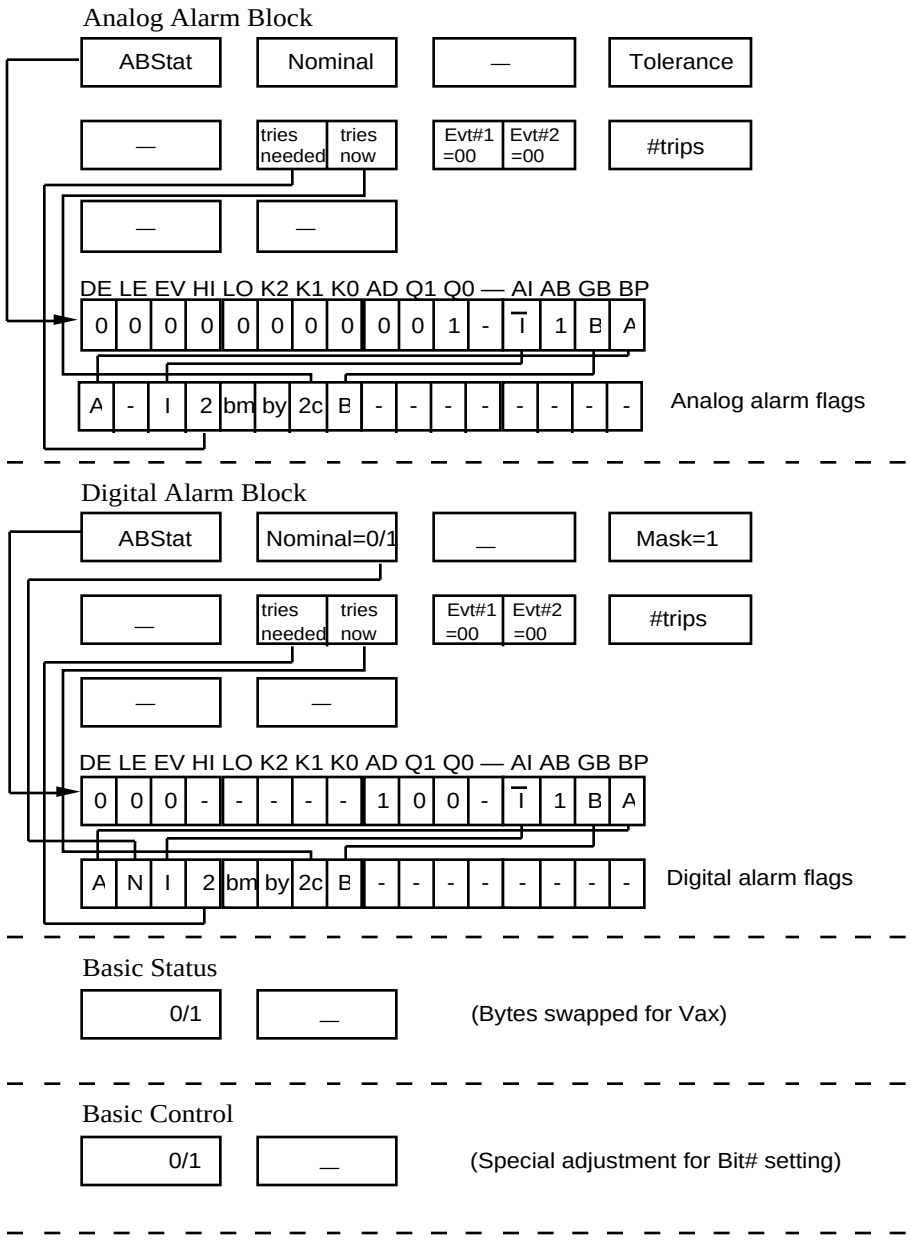
-97 Invalid 9513 timing channel pair
-98 Timing event# out of range.
-99 Invalid data value.
-100 Invalid #bytes of text in Comment alarm control
-101 No DSTRM table of Data Stream queue pointers
-102 Data Stream queue type# out of range
-103 Data Stream queue not initialized
-104 No MMAPS table of memory-mapped board templates
-105 Invalid MMAPS table header
-106 Invalid MMAPS table entry size
-107 Invalid board# for MMAPS table
-108 Invalid directory entry in MMAPS table
-109 End of MMAPS table reached during template processing
-110 Invalid MMAPS command type code
-111 Invalid MMAPS loop params
-112 Invalid MMAPS nested loop
-113 spare
-114 Invalid listype#
-115 Invalid ident type# (error in listype table)
-116 Invalid ident length for this listype
-117 Little console settings switch disabled
-118 Little console external settings switch disabled
-119 Data Server setting not implemented
-120 Invalid listype for this Acnet property

Data format conversions

Special considerations of the Acnet protocol require support of several standard data formats. Logic is included that supports the following standard record structures:

ANALBL	Analog Alarm Block
DGALBL	Digital Alarm Block
BSTATS	Basic Status
BCNTRL	Basic Control
ESTATS	Extended Status

These standard data formats are as follows:



The alarm blocks are the most complex structure to support. The flag word must be edited to conform to the Acnet standard form in response to a data request.

And it must be edited to the Local Station format in response to a setting. The other fields are similarly edited. The tries needed byte may be one or two, according to the 2x bit in the analog or binary alarm flags. The #trips word is returned as extra info in the alarm block. Event-related alarms are not supported.

A special adjustment must be made to accommodate data requests of less than 6 bytes for an analog alarm block. When the read-type routine is invoked to update the answers to such a request, the #bytes requested must be set to at least 6, or the read-type routine will not return the analog alarm flags word that must be edited to make up the ABStat word in the reply. This adjustment also requires that 6 extra bytes be allocated in the answers block (type#9) in order to assure that the extra bytes requested of the read routine cannot be written beyond the end of the block.

For the case of the Basic Status property, the bytes of answers must be swapped to conform to the byte order of the DEC machines. This is also true of some forms of Basic Control, but the data sent with listype #21 (digital I/O via Bit#) is considered a word, where the hi byte is the digital control type# and the lo byte is the pulse delay (when used). So in this case, the bytes should not be swapped.

Limitations of present implementation

Features *not* supported in the initial version of Acnet request handling are the following:

SSDR-related requests
Event-style FTD's
Data offset

It is not intended to support data requests of the "Data Server" type for the Acnet protocol. Idents in a request are *ignored* if they do not include the node# of the local station receiving the request in the first word of the ident. This means that one could send the same request to a group of nodes using the functional group multicast form of network addressing, and each node receiving the request would select out its own idents for answer response. (Obviously the requesting node would need to scan the original request in order to be able to match the answers with the questions.) Currently, however, the Acnet header-based protocols do not permit sending request messages to a group of nodes.

Comparison with "classic" protocol

The Acnet RETDAT/SETDAT protocol for data requests/settings is a very flexible protocol that serves multiple front end computers whose internal software may be organized quite differently. The SSDN component of the request/setting packets is the key that makes it work. The coding of the 8-byte

SSDN structure can be designed for the needs of each front end; neither the console computers nor the central database cares what it is. It only must be correctly entered into the central database.

The “classic” protocol that has been used by the Local Station processors since 1982 is designed to support that particular front end type. The concept of characterizing data requests in terms of *arrays of idents* to be processed in the same manner is used to optimize request update efficiency. Updating an array of channels with analog readings, for example, is distilled down to a 3 instruction loop with the loop count being the number of channels in the array.

This implementation of the RETDAT/SETDAT protocols does *not* rearrange the request into one that can be processed optimally. It can be enhanced at a later date if the extra effort is deemed to be worth the increase in efficiency.

Acnet on UDP/IP

Local Station Implementation

Sat, Jan 23, 1993

General plan

Acnet over UDP/IP is implemented by assigning node#s in the range 0900–09FF to signify the use of UDP rather than raw network communications. Each Local Station front end, since it supports both raw and UDP network transactions, will have two node#s assigned: the raw node# (0614 for example) and a UDP value (0973). These UDP node#s are assigned to each node as needed. Currently, assignments for the 71 installed Local Stations are in the range 0965–09AB. If a Vax console, or any other host that sup ports this scheme, wants to send a message to a 09xx node, it should use UDP encap su lation around the Acnet header-based message to be sent, and it should indicate its own UDP node# in the source node# word of the Acnet header. A database entry for a device that is sourced from a front end that can only speak UDP must specify a 09xx value in the source node# field for that device. Vax DPM software will then use UDP to commu ni cate the request to that front end, using a destination port# of 6801 decimal.

Local Station implementation

Support has been added to the Local Station software for an alternate UDP node#, and UDP-based Acnet messages sent from a Local Station will exhibit the 09xx value in the source node# word of the Acnet header, as described above. Each Local Station knows its own alternate UDP node#, but it does *not* know that of other Local Stations. In addi tion to Acnet support, Classic protocol support has also been extended to use UDP when the target node# is in the UDP range. To derive the IP address for a target node, a "trunk 9" table is used that is simply an array of IP addresses indexed by the xx value from the 09xx node#. Entries are automatically filled by a UDP Acnet message received from a node that specifies 09xx as its source address. It is also filled by any UDP Classic or Acnet message sent from a local station that targets a given 09xx node#. This IP address array is 256 longwords in length. It is currently stored at address $10FA00 in non-volatile memory. The local 09xx value stored at $10507E in the token ring table is sampled at reset time as the local UDP node# for that station.

Server function

Special considerations are needed for the Local Stations because of their built-in server functionality. A request message sent *directly* to a Local Station that includes in its list of idents references to nodes *other* than the receiving node will receive server support. This means that station will accept responsibility for collecting all the requested data from the other nodes for inclusion in its response to the requester. (Note that this fea ture is used for all Linac devices accessed via the Acnet RETDAT system.) To assist in this implementation, a convention is used that requires that the second word of the SSDN four-word record in the Acnet central database device entries also be used for the source node# field for that device. This is necessary so that a serving node can anticipate which nodes will respond to its forwarded request, which is sent via

multicast network addressing in the case that more than one "real" source nodes are involved in the request. For the case of a forwarded UDP request, each reply from a contributing node will have the UDP node# in the source node# of the Acnet header. This is necessary because the serving node must find a match against the node#s in the SSDNs of the original request in order to know how to distribute the reply data into the final reply message that will be returned to the requester.

From the above description of the server logic that interprets contributing node replies, it can be seen that this logic will not work in the case that the list of SSDNs in a request includes some raw node#s and some UDP node#s. This "mixed bag" request is not sup ported. For the case of RETDAT protocol, this situation will be prevented if a simple convention is followed for database entries. For devices from front ends that must be accessed via UDP, the UDP node# must be used both in the SSDN and in the source node# field of the database entry. Thus, Vax DPM will not form a "mixed bag" request.

One node# enough

To help workstation access to data from Local Station nodes, it is desirable to allow idents that use the raw node# to be accessible via UDP, even when the front end supports both raw and UDP. In this case, UDP is used because the *requesting node* has no other option, and the devices given in the database are identified as being sourced from raw node#s. To support this, UDP node#s that match the receiving node's UDP node# are converted to raw node#s when received in a request message, except in the case of the RETDAT protocol request that requires server support. The reason for this exception is that local node references are not separated out for Acnet protocols, as compared with the Classic protocol case. The local node acts as any contributing node.

Installation

The procedure for installation of this new UDP node# support is simple. Before resetting with the new system software installed, install the local node's UDP node# in the non-volatile memory address above. The other part is the IP address table. It can be copied from another Local Station, or it may be downloaded via the ACNAUX function called IPATAB (code=$11) from a Vax. (At this writing, such functionality has not been inclu ded in the Local Station version of ACNAUX, but it can be easily added with logic analo gous to that used to download the physical address table used for trunk zero nodes.)

EACNET Support for Local Stations

Ethernet console access to token ring front ends

Nov 9, 1991

New accelerator consoles cannot connect to token ring, which is the network of choice for the control system front ends, due to hardware limitations. As a stop-gap measure, so-called DACNET server nodes, which are more expensive Vaxes that *do* connect to token ring, have been used to bridge this gap. There are now commercially available bridges that offer much better performance, and such a bridge will be used instead of DACNET servers. Special accommodations are needed for front ends to support the new EACNET, the term used to refer to ethernet-based Acnet.

Console nodes use ethernet addresses in the form AA-00-04-00-nn-E8. The nn is a node number on "trunk 8". Acnet headers include this source node# as 08nn, where here the trunk is shown in the hi byte. For a request message, the source node is found in the 4th word, whereas for a reply message it is in the 3rd word.

When the CrossComm bridge is used, the ethernet source address is altered by having each byte bit-reversed. So a front end on token ring will see ethernet addresses of the form 55-00-20-00-uu-17, where uu represents the bit-reversed version of nn above. In the future, it may be necessary to use ethernet addresses of a different form, in which uu is *not* the bit-reversed version of nn, or in which the last byte of E8 (or 17) is changed to some other value. The value of the first 4 bytes, however, is not expected to change anytime soon.

To support the required address translation, front ends should keep a table of two-byte values, indexed by nn, which can be used to furnish the last two bytes of a destination ethernet address. The table can be built dynamically, or it can be downloaded from OPER using a suitable ACNAUX protocol variant.

To build this lookup table dynamically, a front end should recognize a frame coming from an ethernet node by its source address field. If it looks like the form 55-00-20-00-uu-xx, and targets a node# of 08nn, it can be assumed that it comes from an ethernet node through the bridge. Record the uu-xx bytes in the table indexed by nn. When there is a reply to a node whose node number is 08nn, do a table lookup to get the two bytes to append to the constant 55-00-20-00 to derive the destination network address. Such an address will pass through the bridge to the corresponding ethernet node.

There is a *caveat* regarding frame size, however. Ethernet frames are limited in length to 1500 bytes. The token ring limit is about 4K bytes. If a frame larger than the ethernet limit is to be sent to a console on ethernet, the front end must send it to a special "packeter" node on token ring, which will in turn forward the it to

the ethernet node in packets of less than 1500 bytes. Likewise, when an ethernet console wants to send a large frame to a token ring node, it sends it instead to this packeter node, which assembles the packets together and sends the larger frame onto the token ring destination node. In this case, the front end will receive a frame whose source address is that of the packeter node, not the ethernet node.

A large request message might be sent from a console that results in a short reply message. In this case, the ethernet address may not be seen by the front end, so it cannot reply through the bridge. The solution in this case might be to send the reply through the packeter node anyway. Of course, a previous short frame received from that same console may have resulted in placing a proper entry in the table referred to above, so that this case would not happen. Another option would be to form a default ethernet address from the node#.

Local station front ends automatically pack multiple messages that are destined for the same node and do not exceed the frame size limits into common frames, if they are queued to be sent at the same time. The software is organized to build reply frames at the same time so as to increase the likelihood of this occurrence. This is done to reduce the number of frames transmitted, which can pay big dividends for the receiver in terms of reduced network handling soft ware overhead. Vax consoles should expect to receive frames with multiple messages.

This logic of combining multiple messages into common frames brings up another wrinkle in providing support for EACNET in the front ends. Messages destined for node 08nn cannot be combined if some are larger than the ether net limit and some are smaller, since the real destination node is different. This is easily handled by replacing the target node# of large messages with the packeter node#, without altering the destination node# in the acnet header itself.

The table referred to above in OPER can be obtained by sending a request mes sage using the ACNAUX typecode word of `000E`. The reply consists of 256 words (512 bytes) indexed by nn, where the 00 entry is unused. At this writing, the table contents are 0000, E801, E802,… , E84F, and the rest are 0000. The table should change only rarely, and it can be downloaded at that time if the front end sup ports the typecode (`000F`?) used for downloading this table.

Implementation in the local station software

NetLayer module

In the NetQueue routine, the replacement of the target node# with the packeter node# is done for a message larger than 1496 bytes.

NetInt module

A Node Address Table is part of the TRING table. It has entries for trunk 8 node#s in the range `00` through `EF`. (Entries in the range `F0` through `FF` are reserved for group addressing translation.) This table is cleared when the `TRING` table is manually initialized from scratch, which occurs if the `TRING` table is destroyed. (Resetting the local station does not alter this non-volatile memory table.) Each table entry includes the six-byte network address and a two-byte diagnostic count of the number of frames sent since the last time a frame was received from that trunk 8 node.

The NetRInt token ring receive interrupt routine only allows source network addresses of the form 40020000xxxx and 55002000xxxx. This insures that a reply to a request can be delivered.

The NetSendF routine prepares the token ring destination address. If the target node# is in the trunk 8 range, the network address is taken from the Node Address Table entry indexed by the node# low byte. In the case that the entry is empty, a default address is built of the form `55002000uu17`, where `uu` is the bit-reversed value of the node# low byte.

ANet module

The Acnet Task analyzes a frame received via SAP $68 and dispatches the acnet header-based messages it contains according to the destination task. If the source network address in the frame header is of the form `55002000uu17`, *and* the source node# (found in the acnet header of the first message) is in the trunk 8 range, the six-byte source address is recorded in the Node Address Table entry indexed by the low byte of the source node#, and the entry's count is cleared.

In this first implementation, there is no support for downloading the OPER-based table referred to above, although it could be added in the future if the dynami cally-populating logic proves inadequate.

FTPMAN Implementation Notes

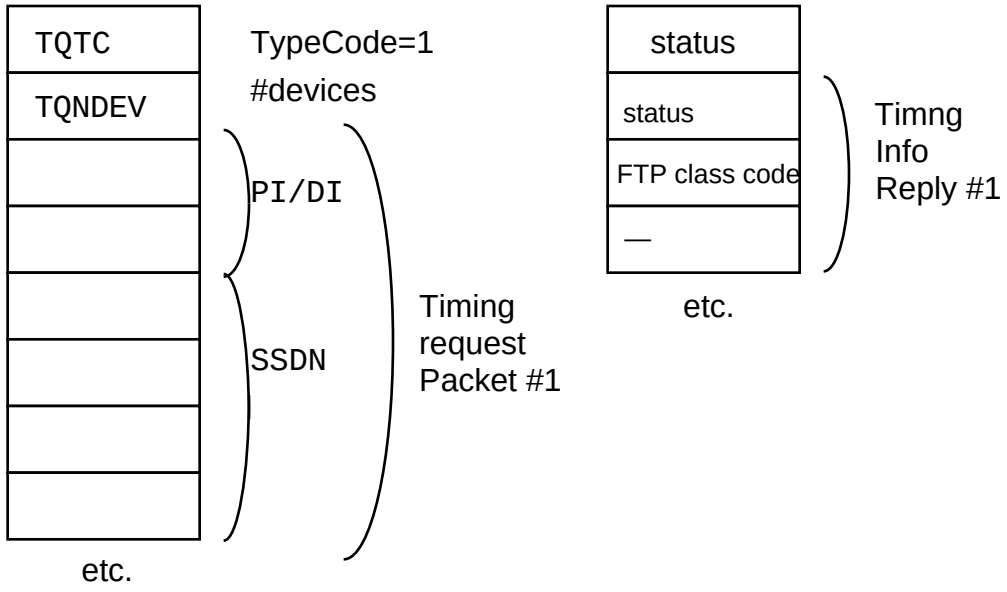
for Linac Local Stations

May 1, 1991

Accelerator data requests are supported by network tasks known by the RAD50 names RETDAT and SETDAT. Data requests for plotting, however, are supported by another network task called FTPMAN. Console plotting of Linac data at 15 Hz is not possible without support for FTPMAN. This note sketches what is required to provide this support for the Linac local stations, which act as front ends to the accelerator Vax consoles. These notes are derived from Jim Smedinghoff's ACNET Design Note No. 49.4 dated March 29, 1988.

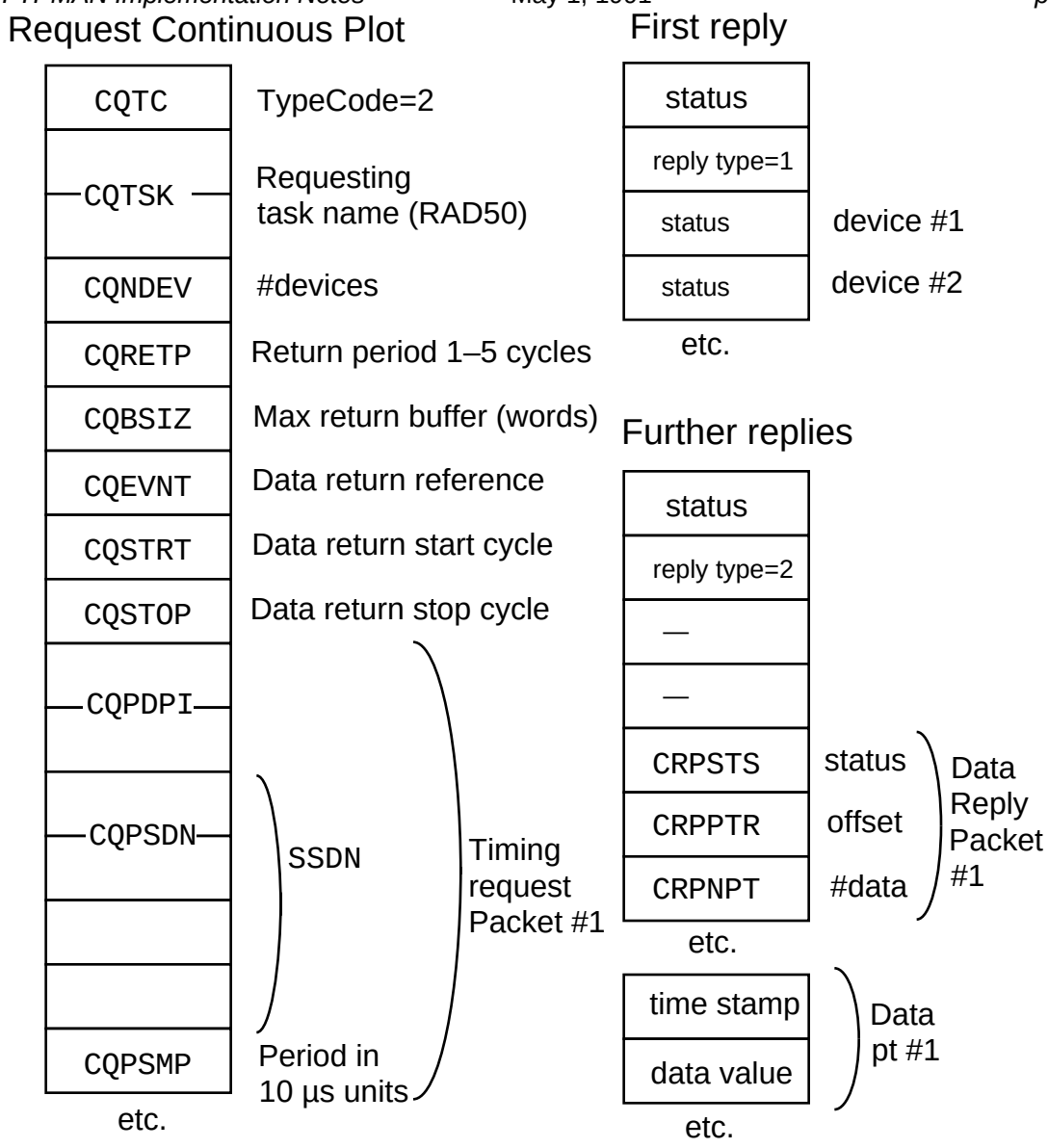
Two types of plotting protocols are supported via FTPMAN, continuous plots and snapshot plots. Plotting 15 Hz data only requires the continuous type.

Activation of plotting requests are done in two stages. The first requests Plot Timing Information, and the second requests the plot data. The first word of the request message is a typecode which distinguishes the request types.



The Linac 15 Hz class code is 5. The #devices in the request is the number of timing request packets. Nothing needs to be remembered about this request for timing info.

To initialize a continuous plot, the protocol looks as follows:



The requesting task name is used to automatically cancel any previously active FTPMAN requests from a given task in a given node. Only one request can be outstanding from a requesting task. The return period can be from one to five 15 Hz cycles, resulting in a frequency of 15 Hz to 3 Hz, respectively. The maximum data return buffer size is influenced by the requester's ability to keep up with the returned data frames. The data return reference word selects clock events to use for sampling the data to be plotted. It will likely be 0, indicating no events (always return data). The start and stop cycle counts require the ability of the front end for noticing clock event #2. The time stamps, in 100 µ s units, also require this ability.

FTPMAN Design for Local Stations

Implementation considerations

Oct 24, 1991

The Fast Time Plot Manager has been implemented for the Local Stations as part of the Linac controls upgrade. It is a “local appli cation” to keep from placing the code into the system itself and to facilitate writing it in a high level language. For optimal integration with the local station system code, however, some special accommodations were added to the system. These notes describe the relevant considerations and the solution implemented.

FTPMAN is a data request protocol. The local stations already supported three data request protocols (Classic, D0 and RETDAT), so this is the fourth. For the new Linac, devices will be entered into the central database as having a common source node—the Server Node ($0601). Because fast time plots will be made of readings or settings of the same devices normally entered on a parameter page, FTPMAN protocol support must include server support, which is automatically provided when an FTPMAN request is received that includes devices from other nodes. The server node then forwards the request to the “real” target nodes (using group addressing if more than one other node is men tioned). These nodes recog nize that the request thus received need *not* be given server support because either all devices in the request are local, or the request was group addressed. The server node receives the replies and builds *its* reply in keeping with the order of the devices given in the original request. It replies to the requesting node according to the period specified in the original request.

For the RETDAT protocol, support consists of two parts. For non-server requests, the Accelerator Task receives network messages directed to RETDAT. A request is initialized and the first reply issued right away. Subsequent replies are issued at Update Task time early in the 15 Hz cycle. For server requests, the Accelerator Task receives the request directed to RETDAT, and it queues the request again to the network either to a single node or to a group address, depending on how many different non-local nodes are represented in the request message. First-time replies from the contributing nodes are sent right away. In turn, the server node’s first reply is sent as soon as these are received. After the reply, during Server Task processing late in the cycle, all accumulated server replies which are due are sent to the requesting nodes.

In order for the Update Task to build non-server replies, or for the Server Task to send out its replies, a linked list of active requests is scanned. All non-server requests and all server requests are in the same linked list. The Update Task only reviews non-server requests, and the Server Task only checks for server requests. In order to support FTPMAN in a manner that integrates well into the system, such requests must get similar treatment.

The reason for wanting to achieve this kind of integration is that this com mon linked list of active requests is the key to providing optimal assembly of network messages into common network frames. The linked list is main tained in an order which is sorted according to requesting node. Therefore, the answer messages generated during Update Task and Server Task pro cessing are queued in an order that is compatible with the network trans mit ting logic, which combines *consecutive* messages destined for the same desti nation node (and using the same 802.2 SAP#) which can fit in a maxi mum size frame.

As a local application, FTPMAN would not, until now, have easy access to such support. Each local application is called as a procedure by the Update Task early in the cycle. But to provide server support, it needs to be called at Server Task time as well in order to deliver server answers at that time. It cannot wait until the Update Task time in the next cycle, because at that time, non-server replies are being returned that may contribute to building server replies.

Generic request protocol support

From analysis of the data request support already completed for both the D0 and accelerator request protocols, there are only a few hooks into the rest of the system needed to provide support for any acnet-header-based request protocol.

The protocol package must be invoked by network messages that are received. This has been done until now by supplying a new *task* that waits for a network message using the `NetCheck` routine that is part of the network layer. As soon as the Acnet Task processes the message, it is placed into a message queue for that task, which was specified in the task's original call to `NetCnct`, on which the task waits via `NetCheck`. For local appli cations, which are *not* tasks, there is only a procedure call interface, so this method will not work. The Update Task calls each local appli cation during data access table processing once per 15 Hz cycle. The following is the new scheme for calling the local application in response to a received message from the network.

The Accelerator Task already waits on a message queue that receives all RETDAT and SETDAT requests, delivered by the Acnet Task. It uses a common message queue to insure that requests and settings are not processed out of sequence. The new scheme expands on this by permitting a local application to request that an additional protocol *also* be received into this same message queue. When the Accelerator Task finds a message which is destined for neither RETDAT nor SETDAT, it searches a new protocol table for a matching entry that contains the protocol's destination task name for requests, such as FTPMAN. From that matching entry, it obtains a pointer to a local application's LATBL entry, which is used to invoke the local application, indicating to it that a network message was received, allowing the local application access its own static variables and other

parameters. In order to pass a reference to the received message, a pointer to the message reference structure of six longwords (including a ptr to the received message in the network frame buffer) can be passed to the local application via the parameter area of its LATBL entry. When the local application has processed the message, it returns to the Accelerator Task, which checks the queue for any more messages.

In addition to providing a connection so that a local application can be notified of network messages directed to it, hooks are also provided that allow it to fulfill requests during Update Task and Server Task processing.

When the Update Task scans the chain of active data requests, it calls a routine to process each one. That routine is given only a pointer to the request block used for that protocol. The Update Task keys on the request memory block type# in order to know what routine to call. It does this for two cases. For newly-received non-server requests, there is an attempt to make the first reply right away. The others will follow according to their request periods. For the accelerator protocol, for example, these routines are called ACUDPNEW and ACUPDCHK.

The Server Task, running late in the cycle, makes a similar scan of the active request chain to decide what routine to call for updating a server request. It also keys on the request block type#. It is necessary to distinguish whether the call to update is for the non-server case from Update or the server case from Server.

When the QMonitor Task determines that an acnet-header message has been completely transmitted, and the optional transmit status has been delivered to the user's variable, it calls a routine based upon the NETQFLG word in the message block. If bit#6 is set in the hi byte of that word, then the pointer to the parent request block, which is at a fixed location within the reply block message structure, is used to obtain the request block type# that, via a search of the protocol table, makes it possible to invoke the update procedure that is part of the local application but separately called. The arguments are a call type# and a pointer to the request block.

One more piece of generic information about a new protocol that is needed is a means of determining the destination node for a reply to a protocol's request. The INSCHAIN routine uses it when it inserts a request into the active request chain, which as noted before, is maintained in requesting node order.

The protocol table entry that is registered by a local application to satisfy the above hooks includes:

Request block type#
Offset to requesting node in request block

Ptr to handler which can be called by Update, Server, or QMonitor

Network task name (such as FTPMAN)

Ptr to LATBL entry that specifies parameters for calling LA.

Pascal interface

The handler registered in the protocol table is declared as follows:

```
CallType= (delChk, updNew, updNServ, updServ);

PROCEDURE Handler(call: CallType; VAR rBlk: ReqBlock);
```

Here `rBlk` is the allocated request block of the type# in the protocol table.

The set of routines for managing the protocol table, found in the OPENPRO module of the system code, are as follows:

```
PROCEDURE InitPro;  {  Initialize protocol table at reset time }

FUNCTION OpenPro(mBType: Integer; rNOff: Integer; Handler: ProcPtr;
                 taskName: Longint; LAEntry: ParamPtr): Integer;
                 {  Place new entry into protocol table }

PROCEDURE ClosePro(mBType: Integer); { Remove entry from table}

FUNCTION HandlerPro(mBType: Integer): ProcPtr; {  Get Handler
address}

FUNCTION RNodePro(mBType: Integer); Integer; {  Get requesting node }

FUNCTION LAEntPro(taskName: Longint); LAEntPtr; {  Get LATBL entry
ptr}
```

Details

The Update Task calls local application FTPM during Data Access Table processing. During the "init" call, static memory is allocated and initialized. FTPM gets the ACRQ message queue id via the `Attach_X` call to pSOS. It calls `NetCnct` to cause messages for FTPMAN to be directed to the same queue waited on by the Accelerator Task. FTPM also installs an entry in the protocol table, providing both a handler (for fulfilling requests and checking for cancelling one-shot requests) and the ptr to the parameters area in the LATBL entry. The handler accepts two arguments: a call type# and a ptr to the request block as found in the active request chain and as described above.

After initialization, the calls to FTPM from the Update Task every cycle may not be useful. However, in order to cancel one-shot requests, it may be convenient to mark the request block to be cancelled by the cycle call to the LA. This is because cancellation may require access to the static variables of the LA, and they are not accessible from the update procedure, although one could consider including a pointer to the static variables area for that purpose in the request block.

Upon termination, if the LA is disabled, the protocol table entry must be cleared using `ClosePro`, and `NetDcnt` must be called to stop queuing further messages into the ACRQ message queue. Finally, the static variables area must be released.

When a network message is received by the Accelerator Task via the ACRQ message queue, and it is neither RETDAT nor SETDAT, the protocol table is scanned for a match on FTPMAN. From the matching entry is obtained the ptr to the LATBL entry which is all that is needed to invoke FTPM. By placing into the parameters area a ptr to the message queue contents just read, FTPM can find the network message and process it in the network frame buffer.

The Update Task, when following the linked list of active requests, calls the handler found in the protocol table entry for the request block type# encoun tered. The handler updates the non-server request, and it should queue an answer message to the network if due. Later in the cycle, the Server Task also follows the active request chain and calls the handler to update and deliver any server-type requests that are due. When the QMonitor Task runs and finds that an Acnet header-based message has just been transmitted, it also calls the handler to check for automatic cancellation of a one-shot request.

In summary, then, the FTPM local application is called by the Update Task during data access table processing each cycle. It is called by the Accelerator Task to process a network message when it arrives. The handler whose entry point is registered in the protocol table is called by the Update, Server and QMonitor Tasks when appropriate to build replies and to delete one-shot requests.

Time-stamps

The first version of this FTPMAN support has a serious problem in that the time -stamps that are recorded with each data point are not correct. It is required of every front end that supports FTPMAN that the reference for the time-stamps be the clock event 02 of the Tevatron clock. This allows the fast time plot program to correlate data points collected from different front ends on the same plot. But the local stations do not yet have access to this clock event signal, which occurs every 5 seconds. There are two solutions to correct for this.

The first is to install hardware to decode this clock event signal that would make it available to a local station. If one local station had this information, it could use the network to send it to all the rest. It might do it only once per minute, rather than every 5 seconds. Local stations can count 15 Hz cycles as well as any node. What is missing is the proper phase. As a result, it might take up to one minute before a newly-reset local station was able to furnish correct time-stamps for fast time plots. A disadvantage of this approach, besides the need to build the hardware, is that it makes one node more important than the rest, which is not in the spirit of distributed systems.

The second solution is to design another FTPMAN protocol variant which would include in the request a suggested starting value for a time-stamp. This solution is a software-only solution. The FTPMAN requester must know about these clock events anyway, because it must interpret the time-stamps to plot the data. But the requester will not know if a front end supports the new variant. So the logic might proceed by trying the current continuous plot standard request format. If an “unsupported typecode” response is returned, it could switch to using the new variant. This might seem messy, but it is unlikely that other front ends that *do* have access to clock event 02 would implement support for the new variant.

The current version of FTPMAN for the local stations is about 1200 lines of Pascal source code that compiles into less than 6K bytes.

FTPMAN Timestamps

Local Station implementation

Nov 11, 1991

The FTPMAN protocol for Fast Time Plots specifies that data points be tagged with a 16-bit unsigned timestamp in units of 100 μ s that is synch ronized with Tevatron clock event 02, the event which occurs every 5 seconds (75 cycles). For 15 Hz data collection for plotting, appropriate for most Linac devices, it is sufficient to keep a cycle counter that is reset to zero on the cycle marked by the 02 event and incremented each cycle thereafter. Its values will thus range from 0 to 74. But Linac local stations do not have easy access to Tevatron clock event signals, at least until recently.

A hardware module was installed at the klystron rf test area in A0 on Nov 4, 1991, that detects a selected set of 16 Tevatron clock events. This will be used to collect spark statistics according to the accelerator cycle on which they occur. It is believed that spark occurrences are not strictly random. If a spark occurs on one cycle, it is less likely to occur on subsequent cycles. One can induce a spark by raising the power, so by programming the power carefully, the spark rate can be made to be much lower on real acceleration cycles than it might be otherwise. One of the 16 events detected by the module is event 02.

For each detected clock event, the hardware module generates a pulse which is stretched until about 40 msec into the 15 Hz cycle. The resulting status bits are read via two bytes of digital I/O through a rack monitor. Watching successive readings of these two bytes, as each event occurs, a one bit appears in its corresponding bit position.

A data access table entry was added to node 062A which sets the reading value of a pseudo-channel to a count of the cycles since event 02 was a “one”. This entry is as follows:

```
1500  03FE  0000  0000
0000  8000  0186  0001
```

Bit 0186 gives the event 02 status, and channel 03FE is the target channel. The 8000 word specifies that the counter be reset when “1” status occurs.

Three other data access table entries are used to send to all Linac local stations a group-addressed setting that targets the 4th word of the Generally Interesting Data area. (The first 3 words are used to receive time-of-day information sent from node 0516 every minute of every day.) These entries are:

```
7F00  0F00  0000  0000
```

```
0000  0000  0000  0000
0D23  0000  0014  3FE0
00FC  0006  0000  0001
0D23  0000  0014  3FE0
062A  0006  0000  0001
```

The first entry establishes a period of about 4 minutes (`$0F00` cycles). The second entry sends a G.I.D. setting to the LINA group address. The third entry sends the same setting to node 062A, as Classic protocol settings are ignored if they are received over the network from the source node.

In each Linac local station which has FTPMAN installed, the FTPM local application is called every 15 Hz cycle. At that time, the timestamp data word is incremented, resetting it to zero when it exceeds 74. When fast time plot data is being updated and tagged by the current time, this timestamp counter is used and scaled to 100 μ s units.

This scheme is not strictly distributed, in that without node 062A, fast time plot timestamp synchronization cannot be achieved. On the other hand, the signals exist at only one node, so this scheme makes the most of it.) The time-of-day clock is similarly multicast by a node more equal than the rest.)

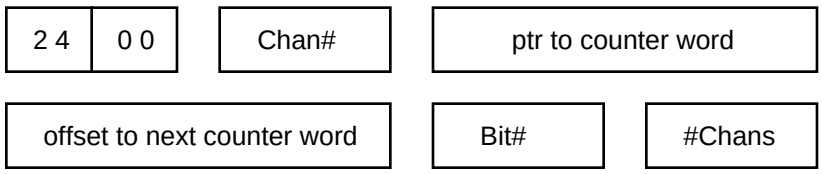
Tests of the behavior of the fast time plot when receiving data points time stamped according to this scheme exhibited purely *forward* time motion, whereas prior to this implementation various degrees of erratic activity of the plotted data points were observed depending upon how nearly to the time of event 02 time was the request initialized.

This scheme should suffice until a new FTPMAN protocol variant is developed which sends a suggested timestamp value along with the request, an imple men tation designed to help front ends use the FTPMAN facility without the need for Tevatron clock signal input.

Data Access Table function
Thu, Feb 18, 1993

This note describes a Data Access Table routine that monitors 16-bit counters to insure that co-processors are functioning normally. It can also monitor the rate of counter advance to show load level. By including a result status bit in the alarm scan, one can generate an alarm message when either a co-processor quits working, or access to a co-processor stops working.

Data Access Table entry layout:



The change in value of the counter word since the last time this entry was processed is stored as the reading of the given Chan# channel. This can be one 15 Hz cycle, or it can be more using an appropriate $7F period entry to specify a sub-multiple rate of execution. For #Chans > 1, the offset longword is used to advance the ptr to get the next counter word address, which is then used to target the next channel.

An optional Bit# word specifies a Bit# that is set or cleared to indicate whether the change difference value is nonzero or not, respectively. (If this option is not used, the Bit# word should be zero.) The reading of this Bit can then be used to generate an alarm message when it is zero, indicating that the counter is not changing. This feature is probably easier than trying to predict what the value of the change should be in order to set the nominal and tolerance values to alarm on the analog channel.

If a bus error occurs when accessing the counter word, the delta value is set to zero, and the (optional) Bit is cleared, indicating that the counter is not changing.

Note that the longword offset value allows accessing multiple co-processor counter words across the vertical interconnect with a single entry, if the counter words are in the same location in each co-processor's memory.

Multiple Networks

Support for both token ring and arcnet

Apr 23, 1990

To support the arcnet connection to the new smart rack monitors to be used in the Linac upgrade project, it is necessary to handle both the token ring network and the arcnet network. This note discusses an approach to simultaneous support of both networks.

The approach is designed to share as much code as possible between the two networks. This means trying to preserve the same protocols for both.

Transmission

A parameter in the message to be transmitted from a local station should indicate which network is to be addressed. Where we have a destination lan-node in the protocol, as for the Acnet header-based protocols, a value of the lan byte can indicate the arcnet network. In the case of the "classic" protocol, the "03" byte can become a lan byte and allow use of a similar scheme.

Suppose there is a lan-network table which gives the network that is to be used for a given lan. The lan# is the index in this table, which should be in non-volatile memory (or in prom). By inspection of the destination lan byte, we get the network to which the message is to be sent.

Classic protocol

The classic protocol presents some special problems because it has no lan byte in the network message. There is a destination node byte which is followed by a byte whose value is always $03. To remain compatible with present systems, the byte should have this value when transmitted on the network.

When a message is received, the destination node byte is changed to the source node, which is obtained from the hardware protocol. The $03 byte is currently changed to the destination node. This last change is made only so that later message processing can determine whether the message was sent in a group-addressed frame. For example, when a data server request is received, a check is made that it was sent to that single node, as group addressing of data server requests is not allowed.

Suppose we modify the logic in this way for the case of the arcnet network. Change the $03 byte to a code which specifies the group address tag in the hi bit (1=broadcast) of the byte. There then remain 7 bits which to be used as the network for the reply. It should have a distinctive value that would not conflict with a lan byte value. An example might be the value $7x, where the

x is the net#, which is not expected to be a valid lan value on token ring. This could allow for 16 networks, which seems to be adequate.

On the transmit side, when a reply is being prepared, the source node of the request is used as the destination node of the reply. Let the $03 byte be used to convey the network to be used in the reply. Suppose it is a copy of the other byte that was received (not including the sign bit). Then a value of $7x could be an indication to use network#x. Other values would be considered as a lan# and looked up in the lan-network table to get the net#.

The lan byte would be used to direct the OUTPQX routine to place a reference to the message into the proper network pointer queue. When arcnet messages are collected into frames, the byte will be altered to the value $03 in the frame buffer so as not to confuse presently installed nodes which require that the byte have that value.

The point is that the $03 byte in the classic protocol is used internally to remember only whether the message was broadcast (or group-addressed). It always has the value of $03 externally. It will now be used also for retaining the lan used for the reply in the classic protocol case on token ring, and it will be used for retaining the net# for the reply for the classic protocol on arcnet.

Acnet header protocol

A special problem occurs with Acnet header-based protocols on arcnet. Since there is no DSAP byte in the frame header, that method cannot be used to distinguish the two protocol types as was done in token ring. So we must determine the protocol type in the arcnet receive interrupt routine by inspection. The first word of a classic protocol message is the message size. The first word of the Acnet header is the flags/msgType word. Since this word does not currently use the hi five bits of the word, we may assign one of these bits for the purpose, say the sign bit. When a frame is sent on arcnet which uses the Acnet header, the code which queues the frame to the arcnet chip should set the sign bit in the first message in the frame. The arcnet receive interrupt routine will look for this bit set (and will clear it) to decide to which message queue the frame reference message should be sent.

OUTPQX

This routine queues a message to the network given by the lan#. For classic protocol messages, the lan# is the “$03” byte. For Acnet messages, it is specified in the destination node-lan word for request/USM messages and in the source node-lan word (from the point of view of the requester) for a reply message. The lan# indexes into the lan-net table to yield a net#. The net# indexes into a NETABLE which contains key parameters of that network.

The present OUTPQX routine has with it a related OUTPQL routine which was used to tag the last entry placed into the OUTPQ with a destination node# for the case of re-issuing requests to sufficiently tardy nodes. This is a bad solution for this multi-network scheme since it is difficult to recover the output ptr queue that was last used, so it is proposed that this one be dropped. The code in the Update Task which makes this call should prepare the destination node byte in the external request block (type#5) *before* calling OUTPQX. Then OUTPQX will properly capture the node-lan and preserve it in the output ptr queue entry for use when messages are collected into frames. The problem for which this is a solution is the case of re-issuing an external request message to multiple tardy nodes at once.

Present lan# situation

With the present system software, the lan byte will turn out to be zero by default, as it has not been used yet. Thus, it may be considered that lan#0 is the default lan. And the default network is the one whose net# is in the lan#0 entry of the lan-net table. By changing the lan-net table, one can make the current software use the any "default" network. Later software will allow the local user to set the lan byte arbitrarily, and that will determine the network to be used.

For the case of replies, the network which delivered the request will presumably always be selected for the reply. The request logic must keep a record of the lan# to use for the reply. To cover the classic protocol case on the arcnet network, there is a means of allowing the receive interrupt routine to specify a net# for the reply, as there no lan# which in the frame header.

NetSend

This routine flushes queued messages to the network. For the case of multiple networks, it is proposed that this call flush all output ptr queues to their respective networks. This call is made after a logical completion of the building and queuing of network messages. It is quite likely that only one output ptr queue is non-empty.

NetXChk

This routine checks whether a message about to be queued to the network is destined for a different node than the first one awaiting to be transmitted. If it is different, then all messages queued to the network are flushed. For the multi-network case, the question is what network output ptr queue to check.

Since this is merely an efficiency issue, in order to more promptly deliver answer response messages packed into frames, we may consider checking only the default network for the time being. An eventual call to NetSend should flush all network output ptr queues.

The point here is that the Network Layer routines tend to hide the concept of the network from the user. The user merely specifies the destination node-lan, and the network used is handled automatically for her.

Multiple Settings

Extension of single setting network message

Apr 13, 1989

Introduction

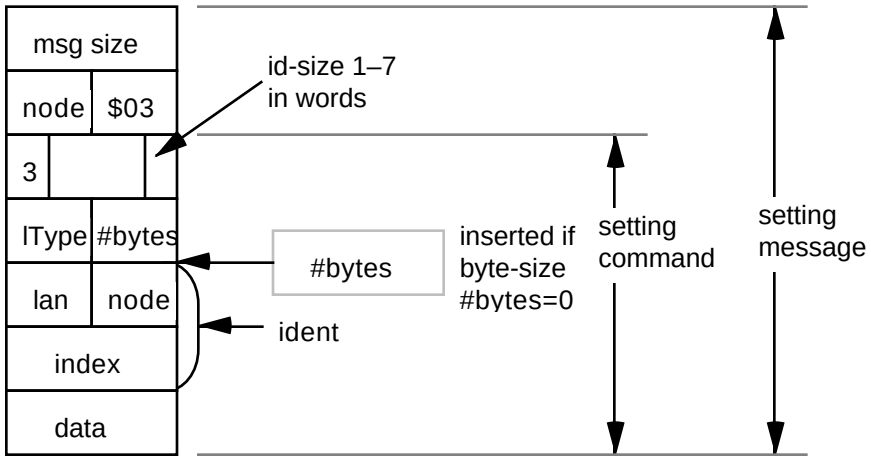
Network setting messages have been limited to a setting of a single listype-ident pair. Although multiple setting messages can be combined into a single network frame, some host-level implementations have found this feature difficult to support. The penalty in terms of network efficiency is quite severe when many settings cannot be combined in this way, as the network overhead time to process a frame is often quite high.

In view of the above state of affairs, it is worth considering allowing multiple setting *commands* to be combined into a single setting *message*. The program which builds the setting commands must realize that they are logically to be executed as a unit. An example of this is the extensive need that D0 has for downloading pedestals and other parameters of its fast physics data acquisition boards. Another example might be using “mults” for simultaneously adjusting steering magnets in a beamline to accomplish a parallel shift in beam position while preserving the beam direction.

It must be noted that the new D0 message protocols certainly allow for this type of multiple settings. However, as they are not yet ready for the VME systems, a small effort in extending the present protocols to support multiple settings should be worthwhile. It is a stopgap measure that allows host-level programming efforts to proceed unimpeded.

Implementation

Recall the present format for a network setting message:



In this example the ident-size is 2 words (4 bytes), and the data is two bytes. Note the definitions of a setting command vis à vis a setting message.

To implement multiple setting commands within a single setting message, merely concatenate the setting commands within the message. The message size word must reflect the total size of all the concatenated setting commands, plus 4 for the setting message header.

Data server option

Now that a setting message can comprise more than a single command, the idents could refer to different lan-nodes in the different commands. If a setting command designates a different station in the ident than the station which receives the setting message, the data server option would allow the setting to be passed on to the designated station. Without the option, such a setting would be ignored.

The data server option is enabled by setting bit#11 (mask=$0800) in the first word of the setting command (the one with the $3000 in it). If a setting message is broadcast, any data server option bit set in the first word of an included setting command will be ignored. This is the analogous treatment of the data server option bit in data request messages.

Network Addressing Notes

What's a node number?

Apr 2, 1990

These notes explore the conventions used in network addressing for the VME Local Stations. There are two sets of network protocols that are handled by the local stations. The "classic" protocols do not use the Acnet header, while both the DZero and Acnet protocols do.

Acnet header protocols—current

When a request is received, the node# byte is taken from the source node/lan word and used to install a copy of the 6-byte network address in the Node Address Table (using the NATENTER routine). This node# is used internally for the destination node for the reply. This method means that any network address can be used for any node, just as long as the requests from different nodes (with different network addresses) do not use the same node#. If they did, the second one's network address would overwrite that of the first one in the NAT. That could mean that the replies from an active request from the first node would all-of-a-sudden be sent to the second node instead. Note that the lan byte is not checked here; however, it is returned in the reply as it was received in the request.

If the local station wants to send a request or USM, it starts with the node# as a destination. The corresponding entry is checked in the NAT. If it is there, then the stored network address is used. If it is not there, then the local station's address is used, with the node# replacing the sixth byte.

Classic protocol—current

There is no source node/lan information in the message. Instead, the sixth byte of the source network address is interpreted as a node#. For any message, the NAT entry is updated. Therefore, the network address used by a node for this protocol is not arbitrary. The last byte must be unique. The other 5 bytes can be used arbitrarily for requesters. But when a local station wants to talk to a node given a node#, it will still assume the network address is remarkably similar to the local station address in the case that the NAT entry is empty.

Acnet console practice

The consoles run by accelerator operators are all downloaded with a common logical node table that contains all the network addresses that are assigned to each logical node. The logical node numbers are 16-bit values found in the database. The values are in the range $0001–03FF, although only the first 72 values are currently used. In some cases, more than one logical node can refer to the same physical node. When a request is received from one of these consoles, the NAT table should be used to keep the network address. Seventy-two entries

would be sufficient for the present.

For front end nodes, the current practice is to hand out blocks of 256 node numbers for specific uses. For these cases, the lan byte of the source lan-node word is greater than three. Thus, the new QPM systems have been assigned the "lan" byte value of 4. The new Linac stations may be assigned the value of 5, for example. DZero stations might be assigned the value of 6, if desired. To simplify the handling of micro-based stations, the lan-node word is used as the last two bytes of the network address, the first four bytes being constant. The value of the constant is $40020000. Therefore, when a request is received, the network address does not have to be saved in a table. Only the source lan-node is saved as the destination lan-node for the reply. When the frame is to be transmitted, the value of the destination network address is simply the constant value for the first 4 bytes and the destination lan-node used for the last 2 bytes.

Local station lan byte

When a local station issues a request of a USM Acnet header-based message, the source lan-node must be filled in for the Acnet header. One way to do this is to use a constant value known to the program. Another way is to keep it in non-volatile memory in the TRING table. Either way, it would be the same value for each request or USM message. (This does not apply for the reply message, since whatever was received in the request is simply echoed in the reply.) This method will mean that the user does not have to enter the lan byte value when she types in a channel number, for example, since it is fixed and supplied automatically.

Lan byte in ident

When a request is received that gives an ident for which data is requested, the lan byte will be a value like 5, for example, if it is a Linac device. The code that checks for a match on the node will have to work for this case. Perhaps the MYNODE word will have to be a value like $0508, for example. The hi byte would be the local station's lan number. Or, the code could perhaps ignore the given lan byte value and only check on the node byte. Right now, the first word of long idents are matched against the whole word stored at MYNODE(A5).

Network Group Addressing

Considerations for standard conventions

8/28/97

Token ring supports group addressing to help filter unwanted messages from specific collections of nodes. New local station software permits increased use of group addressing. There is a need to establish some local conventions for group addressing on the Fermilab token ring network.

Needs of local stations for group addressing

1. Time-of-day message is sent to a functional group address.

2. Destination address for alarms. Alarms are normally sent to a group address so that local stations, and other interested nodes, can receive and display them for local consumption.

3. Data requests using the Classic protocol use group addressing to avoid sending multiple frames when more than one other node is referenced in a single data request.

4. Memory settings may be directed to group addresses using the appropriate internal node# reference. This can permit “gang” program downloading.

Standardized group addressing already selected

1. Time-of-day message sent by a local station that receives the NBS clock signal uses group functional address `C000-4000-0000`.

2. D0 alarms use group functional address `C000-2000-0000`.

3. NWA alarms use group functional address `C000-1000-0000`.

4. IBM software reserves group functional addresses in the range `C000–0000–4000` through `C000-0000-0002`.

5. The token ring chipset uses functional group address `C000-0000-0001` for the ring monitor node.

Local station configuration of group addresses used

There is a table of 16 group addresses stored within the TRING table in non-volatile memory. At reset time, these 16 addresses are copied to another table which is referenced by the network transmit driver using the internal node# values in the range 00F0–00FF. In this way, the system can refer to group addresses as a word value. The meaning of internal node# 00FF is forced to the broadcast network address of C000-FFFF-FFFF.

1. Time-of-day is sent to internal node# 00FE.

2. Alarms are sent to the destination node# in the 4th word of the PAGEM system table (currently based at address 00102000). D0 uses the internal node# 00F0 for this purpose.

3. Data requests, when sent to multiple nodes, are sent to the destination node in the 3rd word of the PAGEM system table.

Token ring chipset support of group addressing

1. All nodes receive all broadcast frames.

2. A node may elect to receive frames addressed to a single "group address." It is specified in the Open Parameter List that is used to open onto the network. This group address is a 31-bit value.

3. A node may elect to receive frames addressed to any set of up to 31 bit-significant "group functional addresses." There are 16 bits available for the "user" to specify so as not to conflict with IBM standard conventions. These user bits are in the range C000-4000-0000 through C000-0000-8000.

Problem looking for a solution

How can we utilize the available choices to support the needs of Fermilab's token ring network?

Group address table

Internal node#	*Address*	*Purpose*
00F0	C000-2000-0000	Alarms destination (D0 example)
00F1		
00F2		
00F3		
00F4		
00F5		
00F6		
00F7		
00F8		
00F9		
00FA	C000-0400-0000	All local stations data requests
00FB		
00FC	C000-CC49-4E41	'LINA' Linac program downloading
00FD	C000-C45A-524F	'DZRO' D0 program downloading
00FE	C000-4000-0000	Time-of-day
00FF	C000-FFFF-FFFF	Pure broadcast

Alarms destination node address depends upon the node's use:

C000-2000-0000	D0 alarms (D0 protocol)
C000-1000-0000	NWA alarms (D0 protocol)
C000-0800-0000	Linac alarms (accelerator protocol)

Values for words 3 and 4 in PAGEM table:

00FA	All local stations data requests (+ name translation)
00F0	Alarms destination

Nodes have optional group address and optional function group addresses:

Assign group addresses to denote downloadable groups.
Assign functional address bits for time-of-day, data requests, alarms.

Local station parameter page

The parameter page on the small consoles uses group addressing in two different ways. One is when a user types a name (6-character accelerator name or 12+4 character D0 name). The other is when there is more than one non-local node represented in the list of channels on the page. This is done so that only a single network message must be sent, even if all 14 lines are of different nodes. The node that is used for either of these cases is the node# in the 3rd word of the PAGEM system table mentioned above.

One might elect to choose to restrict name look-up to a limited number of nodes by careful selection of group addressing. For example, name requests could be sent only to D0 nodes by using the group address given above for D0 node program downloading. This would work for name look-up.

The same group address would also be used when sending data requests to more than one external node. For this it would not work properly. When the data request is sent, if any node does not respond, then another request will be reissued to that specific node(s) that did not respond. It knows which ones did not respond because it examines the list of analog channel idents, each of which includes the node#. This treatment is given repetitive data requests only. One-shot requests are not given this service. For this reason, the analog descriptors (including text, names and engineering units scale factors) may not be collected if the group address used does not address some node(s) in the list.

Also, if a local station is used as a data server node, which collects data on behalf of another node's request, the group address of the data server node will determine which nodes are reachable.

In any case, entering a node:chan on a local station parameter page with no other non-local nodes represented in the list will address the given node directly without using any group addressing. This means that the group addressing cannot prevent access to any given node. It can only restrict name look-ups.

Alarms reporting by local stations

Current alarm generating software in the local station software emits Classic protocol alarms, unless 3 special D0 Device Information Block system tables exist, in which case it emits D0 protocol alarms messages. It does not emit both on the network. When a D0 alarm message is received from the network, a local station converts it into a Classic format alarm message for optional local display.

It is necessary to emit alarms in the accelerator protocol for processing by Aeolus on the accelerator Vax. Some means must be found to select this option for Linac local stations.

Accelerator alarm messages include an EMC (Error Message Code) that must be unique throughout the accelerator system. This is done by including a sub-system byte in the EMC so that different sub-systems cannot conflict. There is an alarm status word, a reading value, and up to 15 words of optional parameters also included in the alarm message.

In order for a local station to be able to receive and display accelerator alarm messages locally, additional information is required that is not needed by Aeolus. It can be included in the optional parameters, as they are not now used by the Linac front end. It is the time-of-day that the alarm occurred, the alarm flags (a variation of the alarm status word), and the analog channel name or binary status bit text or comment text (such as a system reset message).

With the above plan, a local station that receives an accelerator protocol alarm message can convert it into the Classic form for optional local display just as is done for the case of D0 protocol alarm messages.

Network Services

Jun 5, 1989

Introduction

The Acnet network header will be used by D0 for task-to-task communication across the network. The current Acnet services use the concept of a user-specified reply buffer to receive replies to a request message. An alternate scheme is described here.

It is desired to make use of pSOS message queues (also called exchanges) to pass network messages through the system. In this case, the reply buffer is not allocated directly by the user but is allocated by the network software in the form of a circular buffer. A received message is passed to the user via a pointer. This saves the overhead of copying the received message, and it allows for multiple reply messages to be queued up awaiting processing by the requester. Reply messages cannot be overwritten by new replies that arrive before the requester sees them. Furthermore, an AST does not have to be utilized to get around this latter problem.

pSOS Message Queues

Under pSOS, a message queue entry is composed of 4 longwords (16 bytes). The queue is identified by a 4-byte name and also by a 4-byte id. A long message must therefore include a pointer to the *real* message in the message queue entry. Through pSOS service calls, a task can send a message to a queue. It can also wait on the queue for a message to arrive. It does this by reading from the queue; if the queue is not empty, the message at the head of the queue is returned. If the queue is empty, the task may elect to return immediately with status, or it may elect to be suspended until a message arrives, whereupon it will be made ready.

When a queue is created, the queue id is returned for use with most service calls which refer to the queue. If another task wishes to use the queue, it can make an attach call with the queue's name to get the queue id. When the queue is no longer needed, it can be deleted.

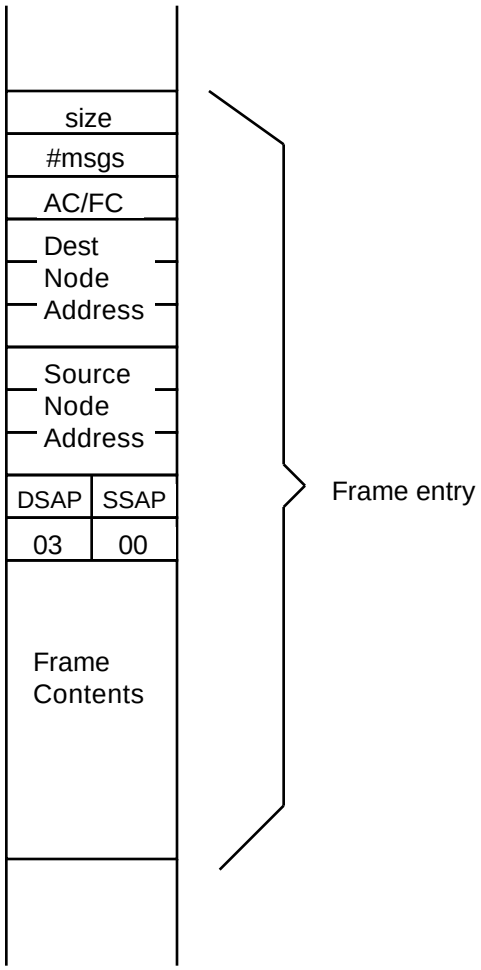
User Network Access

When a task wishes to use the network, it should call NetConnect to connect to the network, giving the taskname and the queue id of a message queue it has created. When a message is received from the network with the proper SAP for Acnet-header messages, it is passed through the message queue for processing.

Software organization for message processing

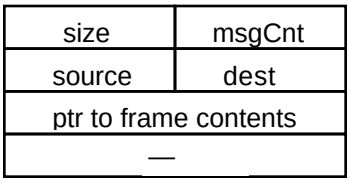
Frames received from the token ring cause an interrupt when they have

been received into system memory. A circular buffer is used to receive network frames. As there is no way to know how long a frame is before it is received, the end of the circular buffer may not get much use. Each frame's entry has the following format:



Everything but the first two words comes by DMA from the token ring chip set. The size word is the total size of the frame entry = frameSize+4. The #msgs word contains a count of the number of messages in the frame. It is decremented by each task which processes the messages. When it is reduced to zero, all messages have been processed in that frame entry, and the space can be made available for additional frame entries. The network receive interrupt must manage the space in the circular buffer, as it pre-pares the buffer pointer and count values used in the receive parameter list.

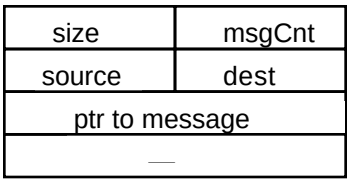
The receive interrupt routine checks for a valid received frame. It examines the AC/FC fields, and it checks for the DSAP and $03 control byte values. Based upon the DSAP value, it sends a frame message to the message queue to be passed to the task which handles the corresponding frame format. The format of this frame message is as follows:



The size value is the size of the frame contents *only*; the length of the frame header has been removed from the received frame size. The source, dest, and msgCnt values are offsets (from the frame contents pointer) to the relevant fields in the frame entry.

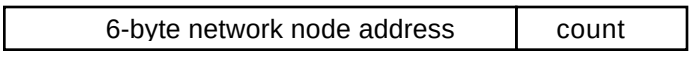
The task which processes these messages (for a given frame format as determined by the DSAP value) must distribute the messages it finds to the appropriate task(s) to handle them. Each frame in general contains multiple messages, and each message header may include a destination task name or source task id that is different from other messages in the same frame. The network hardware delivers frames between nodes; it knows nothing about tasks within nodes.

The frame processing task, such as the Acnet Task, scans the frame contents for messages. Based upon the destination task name (used for request messages and USM's) or the source task id (used for replies), it delivers each message to the message queue appropriate to the designated task. The size of each message is included in the Acnet header in the 9th word. The format of the delivered message queue entry delivered to the designated task is:



The size word is the size of the *message*, including any Acnet header, format block and the message itself. The source, dest, and msgCnt refer to the offsets (from the message pointer) to the relevant fields of the frame entry in the circular buffer.

A table called NADDR is used to keep the 6-byte network node addresses of each node. Internally, each node on the network is denoted by a single byte value. The NADDR table is indexed by this internal node#. Each entry in this table consists of 8 bytes as follows:



When the task handling a particular SAP format receives a request message or a USM (but not a reply message), it checks the entry in this table indexed by source node#. If the source address (from the frame header) matches, the count is incremented. If it does not match, the new source address is entered, and the count is initialized to 1. When a reply message is to be delivered to the source node, the entry in this table is used to prepare the destination network node address for use in the frame header by the network transmit logic. The last few entries of this table contain the group addresses which can be used. The last entry, indexed by node address $FF, denotes the broadcast group address. Other entries, from $FE on down, may be used to denote various functional group addresses.

When one of the message handling tasks, which receives a message pointer queue entry, has processed the message, it should decrement the msgCnt word associated with the frame. This allows the network interrupt routine, when the count reaches zero, to advance the OUT queue pointer to reflect the new space available for more received network frames.

Option Switches in VME System

What do the switches do?

Aug 8, 1989

There are eight option switches included on the front panel of the Crate Utility board. They are rocker switches, where pressing to the right sets the switch=1. They have meaning to the operation of the VME system software as follows:

#7 Used at Reset time to choose to enter the system or a debug monitor.
0= Enter system

#6 If #7=0, this switch can inhibit the automatic restore of D/A and control bit settings.
1= Inhibit settings

#5 spare

#4 Enables output of analog alarms to serial port if #3=1.
1= Enable analog

#3 Enables output of alarm messages to serial port. Analog alarms are included only if switch #4 is set.
1= Enable alarm printing

#2 Enable display of alarm messages on bottom line of little crt. A message is displayed for a minimum of 2 seconds to allow for reading it.
1= Enable alarms display

#1 Inhibits reporting of alarm messages to the network. The alarm scan is always active; this means that alarm status is updated, and trip counts are maintained. The beam inhibit control line, however, is *not* asserted.
1= Inhibit alarm messages

#0 Special TUNECALC closed loop enabled.
1= Tune closed loop enabled

ReqD Notes

Software travelogue

Oct 2, 1990

In order to present a compatible network interface to the accelerator Vax computers, extensive changes are being made in the Local Station software. The Vax assumes the use of a word-size node number, whereas the local station software has always used a byte-size node number. This assumption shows up in numerous parts of the software, and it even affects the page applications as well, as many of them allow input of a node number as a two hex digit field. Support for short idents must be retired, since there is no room for a word size node number in a short ident.

A key module in the system that is severely affected by this change is `ReqD`. This is the routine that a user program calls to make a data request and also the one which supports the data server request, all using the Classic protocol that was designed in 1980 and has evolved somewhat in recent years. This module was written in a style that has since been abandoned, and revisiting the code requires a rewrite in conform to more recently established practices. The previous style involved extensive use of registers with long-term significance, thus avoiding the creation of a local stack frame to contain local variables. It made the program logic extremely difficult to read. Its main virtue has been that it worked. In rewriting this code, a stack frame will be used and the practice of using registers whose significance lasts over pages of code will be resisted. Furthermore, the registers (other than `D0-D1/A0-A1`) will be preserved, thus making it compatible with most hi level language compiler register usage conventions.

The main purpose of these notes is to describe in some detail what the code does for internal documentation purposes. Its use is expected to be limited, however, as there is long term interest in replacing the Classic protocol support with the support for the D0 protocol, as the latter protocol is a logical extension of the Classic protocol and removes many of the limitations of that original data request protocol. The actual retiring of the Classic protocol is not expected for some time, however, as it will affect a number of other users of the local stations from several other platforms. Those users will need to adapt to use of the D0 protocol first. Also, the data server support by the local station will have to be given the D0 protocol before it can supplant the Classic protocol.

Overview

The routine `ReqData` is called by a user program to initiate a data request. The calling sequence is as follows:

```
Procedure ReqData(list: Byte; freq: Byte;
   nLtypes: Byte; VAR listypes: Integer;
   nIdents: Integer; VAR idents: Integer);
```

The `list` is a request-id chosen by the caller that is used to identify the request in the subsequent call to retrieve the data and to cancel the request. Values in the range 0–13 are allowed, depending upon the structure of the `LISTP` system table; it may be modified to extend the range somewhat. See more on this later.

The `freq` byte is actually a period count of 15 Hz cycles to specify a repetitive request. Zero means a one-shot request. The maximum value of 255 is therefore about 17 seconds.

The number of listypes, `nLtypes`, in the range 1–15, specifies the length of the listype array, specified by `listypes`. For each listype and associated #bytes value, the array of idents is processed to produce the resultant answer data. This means that the listypes must be ident-compatible; *i.e.*, they all must use the same ident type. Thus, it is not possible to combine a request for analog channel reading data with memory data, for example, in a single data request. (This is one of the limitations that the D0 protocol removes.) On the other hand, one may ask for readings, settings, nominals and tolerances for the same set of analog channels quite efficiently.

The number of idents in the array `idents` is given by `nIdents`. The hi nibble is used to hold the length of each ident. For example, an analog channel ident is composed of two words, the node# and the channel index, so the ident length in that case is 4 bytes. If the ident length is given as zero, the system assumes a default value. (The current default value is that for a short ident, but that will be changed soon to the value for a long ident.) In the future, the ident length may be required to be specified to open the door to alternate forms of idents for a given listype, such as using a name, for example. Of course, the lengths supported would have to differ to be distinguishable.

Since each ident carries within it a node#, one may make a request for data that refers to a number of different nodes, including the local node. The support software separates out the local idents from the external ones and issues a network request for the external ones. All this use of the network is made transparent to the user. S/he only has to issue the following call to collect the answers:

```
Procedure Collect(list: Byte; VAR status: Integer;
   VAR answers: Integer);
```

The `list` number is the same small value used in the call to ReqData. An error status word is returned, where zero means no errors. The answer data is collected from the local node and/or external node answer buffers that were allocated when the request was initialized. The order and layout of the answer data is specified by the order of the request. The data for all the idents of each listype is separately padded to an even #bytes, in case both the #idents and the #bytes requested/ident were odd.

The system monitors the arrival of external answer fragments that are received from each external node participating in the request. If a node is tardy in returning the answer data, the `Collect` routine attempts to await the return of that node's data using a time-out of about 50 msec past the start of the cycle, after which it returns an error status of either 7 or 8. The value of 8 is used if no answer fragment has been received from that node since the request was initialized and 7 otherwise. If repetitive calls to `Collect` are made, and the tardiness is persistent over 2 seconds, the system reissues the data request to that node, hopeful that the node will revive and begin participation in the request.

To cancel a data request, use the following call:

```
Procedure Delist(list: Byte);
```

Again, the `list` argument is the same small value used in the `ReqData` call. If the request included external nodes, then a cancel message is sent to those nodes to cause them to cease delivery of answer fragments.

Internal list# logic

The `LISTP` table maintains a record of list#s in use, and it provides a means of avoiding reuse of the same list# in a data request for a period of time after a request has been cancelled. This is to prevent misinterpretation of answers to a new request issued immediately following a cancel of a previous request.

The `LISTP` table is divided into a "short set" and a "main set" of entries. The short set is indexed by small values (currently in the range 0–13 as noted above). A short set entry contains a "full list#" that is allocated from the main set. The main set is indexed by a full list#. Its entry contains a pointer to the request memory block (allocated from dynamic memory) that supports the data request while it is active. By maintaining a record of the last-used main set entry and a usage count for each main set entry, and by including some bits of the usage counter in the allocated list#, a newly-freed list# will not be reused for a long time.

It is important that a data request does not use the same list# as one which is already in use. The above `LISTP` logic can provide such service, but the user interface calls do not allow it. Fortunately, the only data requesting programs have heretofore been page applications, and only one of them can be active at one time. If more tasks need to make data requests, it might be necessary to provide user interface calls which *return* the full list# when the request is initialized and which *accept* that full list# in the `Collect` and `Delist` calls. The short set was designed to retain use of the present interface routines but still prevent the above misinterpretation of answers to new requests.

`Delist`

Call `GetListN` to get the full list# associated with the small list# argument. If it is valid, cancel the request by calling `Delist1`, and clear the short set entry by calling `SetListN`; otherwise, simply return.

`Delist1`

Call `GetListP` to get a pointer to the request memory block. Clear the main set entry by calling `SetListP`. Delete the request from the chain of active data requests. Capture the pointers to any external request block and/or a total answers block. Release the allocated request block memory. If there was an external request block, queue a cancel message to the network using the same destination node (which could have been a multi-cast address) that was used in making the external data request originally, and release the external request block. If there was a total answers block, release it also. `Delist1` preserves all registers. Its single argument is the full list# in D0.

Data Server requests

A data server request is one which originates from another node on the network. When a data request message is received that specifies the use of the data server, it is supported by the system on behalf of the network requester. The Server Task, which runs every cycle at about 40 msec into the cycle, scans the chain of active data requests for data server requests, calls `Collect` to retrieve the data (without waiting), and it queues the resulting “total answers” to the network requesting node. A total answers memory block, referenced by a field in the request block, carries the answer response to the network.

`ReqDataS`

This entry point is used by the Network Task when it receives a data server request. If the full list# (specified in the network request by the requesting node) is the same as one found in the total answers block of a currently-active data server request, then that request is cancelled. A new list# is obtained via `NewListN`. Note that the requesting node’s list# is not used, since we cannot guarantee that it would not conflict with a currently-active

LISTP entry. However, the original list# is retained for inclusion in the total answers response to the requesting node.

ReqData

The call is converted into a ReqDataS-compatible call by appending an additional zero argument to stand for the requesting node# argument that is the last argument placed on the stack for ReqDataS. The small list# argument is converted into a full list# via GetListN. If it is active, then the short set entry is cleared via SetListN. A new list# is obtained via NewListN, and is recorded in the short set entry via SetListN.

ReqdCom

This code is common to both ReqData and ReqDataS. Arguments nLtypes and nIdents are checked against reasonable ranges. The listypes are checked for being ident-compatible.

The number of bytes needed for an external request block and for the main request block is evaluated by scanning the idents for the number of idents from each external node represented in the request. Memory is allocated both for the request block and for an external request block, if needed, and the blocks initialized.

The pointer-type routines are called for each listype to translate each local ident into an internal pointer and each external ident into a reference to the predictable part of the external node's external answer buffer where the answers will be placed when the answer fragment message is received from that external node. The result of this "compilation" is an array of "internal pointers" that are interpreted at data request fulfillment time (via Collect) by read-type routines. This interpretation loop is optimized for speed, as data request fulfillment may be done for a number of active requests at 15 Hz.

The "Age" and "Cntr" fields in the external answer pointers are initialized for detection of tardy external nodes. The main set list# is established via SetListP.

If the request was a data server request, a total answers block is allocated and initialized for later queuing of the total answers to the original requester.

The new request is connected into the chain of active data requests via InsChain, which places it adjacent to another active data request from the same node, if there is any.

If there is an external request block, it is queued to the network via OUTPQX.

Note that ownership of each of the three memory blocks is assigned to QMonitor Task via `Assign`, as QMonitor may likely be the one which may have to release the memory when the request is cancelled. If the user calls `Delist` to cancel a request, a check is made to see whether the external memory block or the total answers memory block has been queued to the network but not actually yet transmitted. In that case, a flag is set for QMonitor to free the memory when the message has been transmitted. Otherwise, the ownership of those blocks is reacquired via `Grab` so that the blocks can be freed immediately.

UDP Layer

Support routines

8/28/97

For support of applications that communicate via UDP/IP, some routines analogous to those of the Network Layer are provided. These routines manage the port assignments in the same way that the Network Layer routines manage the task name assignments. (Note that UDP communication that uses the Acnet or Classic protocols does not need to use these routines. They are automatically managed using the Network Layer; in a sense, the Network Layer routines operate at a higher level.) The UDP routines are designed to be used by the server programs that accept UDP datagrams containing either Acnet or Classic protocol messages. They can also be used by user applications that want to communicate UDP datagrams that contain other protocols.

```
UDPCnct(portReq: Integer; qId: Longint; evtMask: Integer;
        VAR portId: Integer): Integer;
```

The first argument of this function is a UDP port# to be assigned. If it is zero on entry, a dynamic assignment is requested, and the newly-assigned index is written to the `portId` variable, unless none is available. If the value for `portReq` on entry is nonzero, it is a request for assignment of the given port#. If the given port# is already assigned, `UDPDcnt` is first called to close the previous connection.

The `qId` is the message queue used for receiving messages directed to the assigned port#. The `evtMask`, if nonzero, specifies the bit mask used to signal the calling task of the arrival of a message.

```
UDPDcnt(portId: Integer): Integer;
```

Remove the assignment of the given port# from the port table. The message queue field is cleared to indicate that the entry is available.

```
UDPQueue(portId: Integer; VAR mBlk: MBlkType;
         VAR xmitStat: Integer):Integer;
```

Queue the message contained in the message block to the network and flush the network queue. The message block is a special type $0016 used for UDP messages. NetXmit will precede the message with the IP and UDP headers formed from the destination node# within the message block.

```
UDPCheck(portId: Integer; timeOut: Longint;
         VAR mRef: MRefType): Integer;
```

Check for a message waiting in message queue used by the given portId. A

timeOut value of –1 specifies "no wait." A positive value is in 100 Hz ticks.

```
UDPRecv(VAR mRef: MRefType; VAR buf: BufType; maxSize:Integer): Integer;
```
Copy the message referenced by the mRef structure into the given buffer.

```
UDPOpen(port: Integer): Integer;
```

Establish an entry for given port# in the port table. A message queue is automatically created using the given port#. If the argument is zero, a dynamic assignment is made. The value of the function result is the portId. In this case, a zero result is an error, as the portId should be positive.

```
UDPClose(portId: Integer): Integer;
```
Close UDP port indicated by portId. This calls UDPDcnt and also deletes the message queue used by the connected port.

```
UDPRead(portId: Integer; VAR buf: BufType; mxSize: Integer;
        VAR mSize: Integer; VAR srcN: Integer): Integer;
```
Check the message queue used by the given portId. If a message is present, return a copy of the message, along with its size and source node#.

```
UDPWrite(portId: Integer; VAR buf: BufType; size: Integer;
         node:Integer; VAR xmitStat: Integer): Integer;
```
Write the message to the network. The function NetQueue is called after preparing the message block using the buffer contents. The node# is assigned from InsIPARP and specifies the info needed to fill the IP and UDP headers.

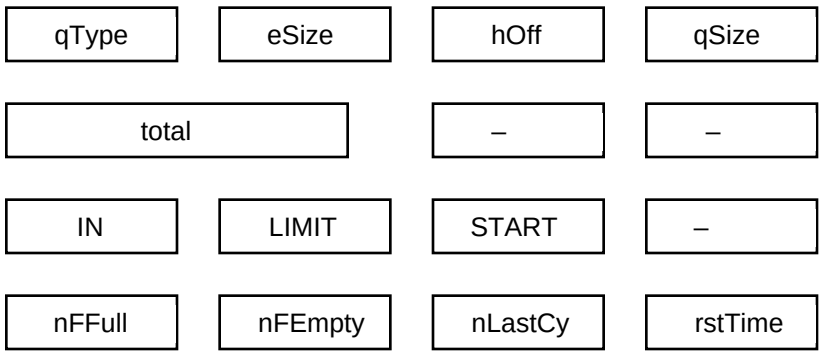
Clock Event Queue

For Clock Decoder board

Sep 6, 1989

The Clock Decoder board captures clock events and records them along with a 720 Hz time-stamp into a hardware fifo. In order to make this clock event data accessible to multiple users, the hardware fifo is read by a routine invoked from a Data Access Table entry, and its data is copied into a software circular buffer. This Clock Event Queue is implemented as a data stream and is therefore accessible via data requests using the data stream listypes.

The Clock Event Queue has the following format:

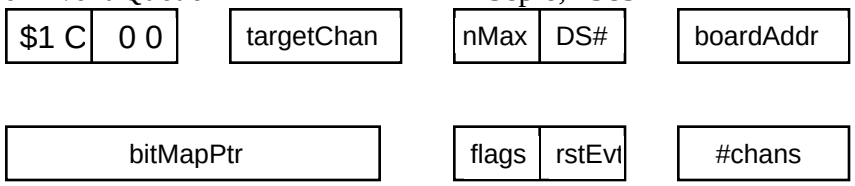


This layout is excerpted from the Data Stream Implementation document. The `qType`=1, the only currently supported queue type. The `eSize`=4, as the packets of data consist of a clock event# word followed by a time-stamp word. The `hOff`=24, the offset to the last 4 words which are clock event-specific data. The `qSize` is the total space used for the Clock Event Queue. The `total` long-word is a count of the total number of packets ever written into the queue.

The `IN`, `LIMIT`, and `START` words are used by the queue management routines associated with queue type 1. The `IN` word is the offset to the next available space in the queue. The `LIMIT` is the same as the queue size. The `START` word is the offset to the first packet in the queue. When the advancing `IN` word reaches `LIMIT`, it is reset to `START`.

The `nFFull` word is the number of times the fifo has been found with fifo full status. When this happens, the fifo is cleared. The `nFEmpty` word is the number of times the fifo has been found to be empty at the start of processing. The `nLastCy` word is the number of events that were found the last cycle that the fifo was accessed. The `rstTime` is the time-stamp value that was last read at the occurrence of a cycle reset event. It is used to adjust the time-stamps read to be relative to the time of cycle reset.

The Data Access Table entry format is as follows:



There are three jobs which can be done by this entry with the clock event data. The Clock Event Queue (as a data stream) records packets about each event that is read. A range of analog channels is updated with the most recent time-stamps for the corresponding events. A bit-map is updated with bits set to indicate the occurrence of the same range of clock events.

The `targetChan` is the base channel number of the range used to hold the time-stamp data, likely with the option enabled to store these time-stamp data relative to the time of the cycle reset event. The `nMax` byte is the maximum number of times the fifo is read while processing this DAT entry. The `DS#` byte is the data stream index used to identify the Clock Event Queue data stream according to the `DSTRM` table. The `boardAddr` is a word that gives the Clock Decoder board's fifo address in VME Short I/O space.

The `bitMapPtr` is a pointer to the base of the bit-map array which contains bits set to 1 whenever the corresponding event occurs. The bit-map may optionally be cleared at cycle reset time. The `flags` byte includes the following options:

bit#	option
7	1= enable cycle reset event logic
6	1= enable bit-map updating
5	1= enable bit-map clear upon cycle reset
4	1= longword bit-map array, 0= byte bit-map array
3	spare
2	spare
1	spare
0	spare

The cycle reset event# is specified by the next byte. It is only used if the cycle reset event logic is enabled. The `nChans` word is the number of channels starting at `targetChan` that are used to record the corresponding time-stamps. Events are processed in the range 00–n, where n= `nChans-1`. It is also used to give the range of events that are recorded in the bit-map table, if the bit-map is enabled.

VME Clock Timer Board

Analog control of AMD9513 channels and clock events

R. Goodwin

Oct 10, 1989

The VME clock timer board uses four AMD9513 chips each of which provides two 32-bit timers with one μ sec resolution in the settable delays. It also includes a 256-byte memory which specifies which of the channels (up to eight) is triggered for each of the 256 possible clock events. Each bit position in the byte corresponds to a different timing channel. This note describes how the VME software implements control and readout of these timing channels.

Design

The generic parameter page is typically used for display and control of numeric values via analog channels. When new devices are added to the system, it is worth trying to see if they can fit into this generic scheme and thus inherit the features that parameter page interaction provides. It also serves to present a consistent interface to the user. The features of the clock timer board that require support for numeric values are these:

- Read and write the 32-bit delay settings (timing channels)
- Set possible multiple clock events which can trigger the delays
- Read what clock events are currently set for each timing channel

Since the delay values are 32-bit quantities, and the VME software does not support 32-bit setting values, the timer delay channels are represented by a pair of analog channels which include a “coarse” and a “fine” control. The coarse channel is in units of msec, and the fine channel is in units of μ sec. On a parameter page, it is easy to read out this way, especially if the engineering units are chosen as msec for both coarse and fine channels. For timing to only msec precision, adjustment of the coarse channel is sufficient. Fine tuning is possible down to the hardware resolution by use of the fine control channel. This coarse and fine treatment also helps to solve the knob resolution problem inherent with adjustment of hi resolution devices.

The use of coarse and fine channels as described above allows for more than one pair of coarse and fine values that can result in the same time delay. Since the counters can be read back to provide a reading of the delay value, and since that conversion into coarse and fine values in *not* ambiguous, the setting ambiguity is resolved by allowing the fine channel to carry into or borrow from the coarse channel when the end of its range is reached.

With the fine channel using μ sec and the coarse channel using msec, the limit in the range of delay setting is only about 32 seconds (if we keep the values positive). If this is insufficient for the users’ needs, then the coarse channel could be changed to have units of 10 msec, which would allow for a range of delay

setting of about 5 minutes.

For setting multiple events, a set of analog channels is needed each of whose setting values is a clock event number. The number of such channels to be installed can be tailored to the needs of the users based upon the intended use of the timing channel and on how the entire clock system is organized. When a clock event setting is made to one of these channels, it checks to see whether that event is already selected for that timing channel. If it is, the setting is ignored. If it is not, the current event setting of the channel to be set is first deselected, and then the new event is selected. This insures that all such setting channels will have unique event number setting values. Note that this logic means that one can adjust an event number with the knob and end up with only the last clock event selected. A zero value setting can be used to deselect an event that was set before. But since there is actually a clock event #0, the value 256 can be used to cause event#0 to be set.

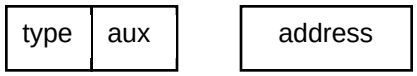
As a shorthand means of clearing selected clock events, a special setting is allowed that causes clearing of all selected events from the 256-byte RAM for a given timing channel. At system reset time, when the restore of D / A settings is taking place in channel number order, the channels holding clock event values will reselect those events for a given timing channel. This means that a *lower-numbered* channel should be used for the purpose of clearing all selected events for that timing channel.

To provide a readout of the selected events that were set, one could merely copy the setting value to the reading. This would partly work, but it has difficulty with the “clear all” setting just described. To get around this, a reading of these event channels supplies a copy of the last setting only as long as the corresponding RAM bit remains set. As soon as it is cleared, by whatever reason, the event channel will have its setting cleared, and the reading will then become zero also.

The readout for a “clear all” channel is a count of the number of clock events selected for a given timing channel. This channel could be scanned for alarms to insure that the number of selected events did not change unexpectedly. To execute the “clear all” function, a zero value must be used for the setting. Then the reading would naturally be expected to become zero as well.

Analog control field

The specification of the control parameters needed for 9513 channel control must be contained in the Analog Control field of the analog descriptor. The format of that 4-byte field is as follows:



The first byte indicates the type of analog control, which in this case would use the value `$0F` to specify 9513 timer type. The meaning of the other three bytes is dependent upon the type, so we are free to choose anything convenient. Let the aux byte specify which timer channel, whether it is the coarse or fine word, and whether it is a delay setting, an event setting or a clear-all-events setting. The address word is enough to indicate the board's base address as it is presumed to be in VME short-I/O space, which means that the upper part of the address is $FFFF. (In fact, the base address must be on a 4K boundary, which means that only 4 bits is really needed to specify the board's VME base address.)

Specific values of the aux byte are:

`$0x` Coarse delay chan #`x`
`$1x` Fine delay chan #`x`
`$2x` Set event# for chan #`x`
`$3x` Clear all events for chan #`x`

The value of the x nibble is 0–1 for the prototype board. It will range from 0–7 for the final board, since that board supports 8 timing channels. Each timing channel will require four analog channels (or more, in order to handle multiple selected events) to cover all choices.

The following steps are performed to set the two timer channels on the prototype board:

Chan #0	Chan #1
$C100 → (17)	$C100 → (17)
$01E1 → (01)	$03E1 → (03)
$1221 → (02)	$1421 → (04)
$0000 → (09)	$0000 → (0B)
lsw → (11)	lsw → (13)
msw* → (0A)	msw* → (0C)
(63) (6C)	

The numbers in parentheses are register selects at the byte at offset $810 from the board's base address, followed by an access to the ls byte and then the ms byte to make up the word, each referencing the byte at offset $800 from the base address.

For the final board, there are four 9513 chips used to support a total of 8 timing

channels. Successive 9513 chips are addressed with 32 bytes of address space each. The specification parameters are the same as above for each pair of timing channels supported by a 9513 chip.

When a setting is made to a delay channel, the entire initialization logic will be performed the first time a delay is written. The coarse and fine channels should be arranged to be consecutive channels so the setting software can put the two halves together.

The lsw and msw* refer to the lo order and modified hi order words of the 32-bit setting. The last step does the Load and Arm operation. If the Master Mode register is read and found to not be $C100, the entire chip must be initialized. If the a timing channel's mode register is read and has the value not ending in $E1, then that counter must be initialized as above. If the Master Mode register looks good, and the counter's mode register also looks good, the only the lsw and then the msw* words should be written to set the counter delay. The other steps should be skipped.

Internals

A new analog setting routine called `SET9513` was written to support all settings for this board. It is invoked when the analog control type byte=`$0F`. It uses the aux byte and the addr word to do its I/O, including initializing the 9513 chip when necessary.

Three new data access table routines were written to support readings of the types mentioned here. Type `$17` invokes `RDEVCNT` to tally the number of clock events selected for a given set of timing channels on one board. Type `$18` invokes `RDEVNTS` to test whether the last selected event is still selected in the trigger RAM. Type `$19` invokes `RD9513D` to read back the coarse and fine delay counts.

All four new routines are contained in the `MOD9513` module of 500 lines of code which assembles to about 900 bytes.

Event-driven Replies to Data Requests

In sync with clock events

Wed, Mar 23, 1994

With the addition of clock event detection hardware in the digital IndustryPack board used in IRMs, we can provide event-driven replies to a data request. For the Classic Protocol, one must be able to specify what clock event should be used to indicate on which 15 Hz cycles the data should be sampled. Using this facility for Linac, for example, one could then reply to a data request only on beam cycles.

The Classic Protocol format for data requests includes the following:

period	#ltypes

The period byte is expressed in cycles (15 Hz), allowing for any period from 1–255 cycles, or 0 for a one-shot request. For event-driven replies, an 8-bit clock event# is needed, so it is natural to specify this in place of the period byte. But then we need to mark the fact that the “period” byte is really an event#. The #listypes byte is usually limited to 4 bits or so, as a Classic request is for a matrix of data to be returned, with all idents processed for each listype#. If 4 bits is enough space for the #listypes field, we can use the upper 4 bits of that byte to contain flags, one of which can mean that the “period” byte is really a clock event#.

When the specified event occurs, an update of the request is generated, so that the data from the data pool is returned on that 15 Hz cycle in which the event has been detected. How can the logic recognize which events have occurred?

The event detecting hardware is programmed to generate an interrupt whenever any event is detected. The interrupt routine reads the event from a FIFO, allowing for many events to occur almost simultaneously without being lost, time-stamps it, and writes the time-stamp into the clock event times table. The information about clock events is present in this table, but it is not so easy to process it in order to quickly check whether it is time to reply to a data request.

One possibility is to maintain a clock event queue, in which is recorded the event# for each event that occurs. The interrupt routine, besides updating the clock event times table, would also write into this queue. To make it easy for a host to read out the contents of this queue, it can be designed as a data stream. But DSWrite, normally used to write records into a data stream, has too much overhead processing for interrupt code, so we can access this queue directly more efficiently.

Now the process of deciding what events have occurred since the last request update is easy. For each request, there must be kept an OUT offset into the event queue that indicates the next record to be checked. Scan all event records written

into the queue since the last update, looking for a match on the specified clock event#. If there is a match, then it is time to update and return a reply to the request.

Another approach is to maintain a bit map of events that have occurred since the last cycle. For such requests, logic must be done each cycle to determine whether it is the time to reply. As a result, replies may be updated up to 15 Hz, in the case that the event occurs at 15 Hz. To maintain such a bit map, one must be careful, as events are processed by an interrupt, and the bit map may change between execution of any two instructions. The task-level solution for this is to copy the bit map into another area, then exclusive-OR the copied bit map into the dynamic one. In this way, any bits that were set via interrupts occurring since the bit map was copied are not lost. They will be detected on the next cycle.

A requester of event data could read out the second area at 15 Hz, thus insuring that all events would be noticed. A request slower than 15 Hz would not be able to detect all events, of course, by simply looking at this copied data. If the requester didn't care about 15 Hz events, but only about Main Ring reset events, say, a new listype ould be designed that gave the bit map at a slower rate, but processing would have to be done at 15 Hz to inclusive-OR the 15 Hz samples of events during those cycles between updates. This may be difficult.

Implementation

Support for event-driven replies has been implemented for Classic Protocol data requests, using the bit map approach internally to detect whether a given event has occurred since the last cycle. In the case that no events of the given type occur, no reply will be generated. This makes it difficult to be sure that the data request was received. The same is true for a server node; it cannot report that a reply is tardy, unless it also knows about the event, too. At the moment, this situation is not detected, but the server node does need to detect the events in order to send replies to the requester.

Generalization

It may be useful to generalize the meaning of events to include conditions that are not actually clock events. By setting one of the bits in the bit map that is not used as an actual clock event according to a condition detected by Data Access Table processing or by a local application. But the need for the server node to know about the special condition means it might be better if the server didn't have to know about events itself.

condition means it might be better if the server didn't have to know about events itself.

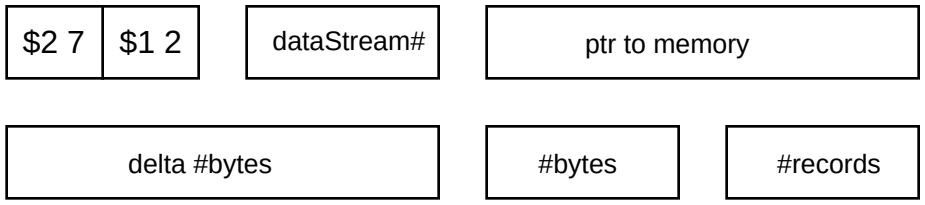
Memory Data Streams

Data access table entry type

Dec 22, 1991

Data streams have been a part of the local station system software for a few years. The original use was to collect clock events from special hardware for the Loma Linda control system. Since then, major use of data streams has been made with the D0 high voltage control system. In that case, a separate processor writes its data into a data stream for which flexible access is sup ported through three dif ferent listypes. This note describes a data access table entry type which pro vides for *copying memory data into a data stream*. Access to such data can be made by any number of users, a fundamental feature of data stream implementation.

Data access table (DAT) entry format



The entry type# is $27. The DSTRM table# is $12. The dataStream# is the DSTRM table entry#. In that entry is described the type of data stream queue, its entry size, total size and base address, and an 8-character name as a diagnostic.

This DAT entry provides for copying data records into the data stream from areas of memory. Beginning at “ptr to memory”, “#bytes” are copied into the specified data stream. If #records > 1, then additional records of the same size are copied, applying the “delta #bytes” to the “ptr to memory” each time.

The data actually written into the record is the memory data preceded by 16 bytes of header information in the following format:

time-of-day	8-byte BCD
ptr to memory data	4-byte address
spare longword	4-byte zero

As always, the first 4 bytes of the response data are two words signifying the number of records included in the reply and the entry size of each record. If the entry size is zero, the first word of each record is the record’s size.

Alarms Task

Local Station Software Module ALARMS

R. Goodwin

Aug 9, 1989

Function

The Alarms Task performs the alarm scanning of analog channel readings and binary status bits every 15 Hz cycle. If a channel reading is out-of-tolerance, or if a status bit value has the wrong state, its alarm state is "bad." Changes in alarm state are reported by messages sent to an alarm destination node. At the conclusion of the alarm scan, if any channel or bit is in the "bad" state that has the inhibit bit set in its associated alarm flags, an inhibit control line is asserted. This may be used to inhibit beam when key devices are down.

Task events

Event #0 is used to signal that the alarm scan is to be performed. This event is sent by the interrupt routine that runs in response to the 15 Hz interrupt. (The external 15 Hz trigger pulse plugs into a Lemo connector on the Crate Utility Board.) This alarm scan is scheduled to run each cycle after new data has been read and network data requests have been fulfilled.

Event #3 is used to signal invocation of the closed loop code. It is sent by the alarm scan code at 15 Hz; hence, closed loop code runs after the alarm scan finishes. Closed loops can be run at other times that this event is sent to the alarms task. The operation of a closed loop may require this if, say, a closed loop starts an action that results in an interrupt that requires post-processing to be done by the closed loop logic. *Closed loop logic is not used in the Loma Linda system.* The `CLOOPDMY` stub module satisfies the linker.

During the alarm scan, if any alarm messages were queued to the network, event #3 is sent to the Update Task to flush the queued messages to the network.

Option switches

Some of the option switches on the Crate Utility board's front panel influence the handling of alarm messages. The most significant of these is option switch bit#1, the alarm inhibit switch. When it is on, the alarm scan is performed as usual, but alarm messages are *not sent to the network*. They can be displayed locally, however. The inhibit control line is not asserted while this switch is set, in order to prevent having the inhibit line set while sending no network message to say why it is set.

Local display of the alarm messages is handled by the QMonitor Task, as it is the one which "cleans up" after messages that are sent to the network. All alarm messages that result from the alarm scan are queued through `OUTPQ`. But when the alarm inhibit option switch is set, a "used" bit is set that prevents actually sending the message to the network. But QMonitor sees the message as it processes the `OUTPQ` entries and can process such messages also. To enable the local display of alarm messages, set option switch #2 to enable display of the messages on the bottom line of the small screen display, and/or set option switch #3 to enable output of encoded alarm messages to the local serial port. Also, option switch #4 must be set to include analog alarm messages in either type of local display. See "Local Console Alarms Display" document for more information about the "bottom line" alarm message display.

Alarms Task initialization

Various pointers are pre-computed for more efficient processing during the alarm scans. Alarms scans are inhibited for about 1 second following system reset. The "reset alarms" control bit described below is set to force all "bad" alarm messages to be produced upon the first alarm scan.

Alarm scan

There are two special Bits of status that are used by the alarms task. They are Bits $A0 and $A1 in every node. At the beginning of an alarm scan, each of these special Bits is checked for special processing.

Bit $A0 is set by a host to clear the analog and binary trip counts for all channels and Bits in a node. (A trip count is the count of the number of times a channel or Bit has made an alarm state transition from good to bad.) Typically, a setting command is sent to all nodes via a broadcast setting after producing a report of all the trip counts of interest in every node. The bit is self clearing.

Bit $A1 is set by a host to "reset" the alarms in a node. The bit is set upon system reset by Alarms Task initialization. When the bit is found set, the good/bad status bit in the alarm flags is cleared for all channels and Bits. The result of this is that any channel or bit in alarm will, on the next alarm scan, cause an alarm message to be generated. The bit is self clearing.

Alarm Flags

The format of the alarm flags byte used for either channels or Bits is:

Bit#	Meaning
7	Active. "1": this channel or bit should be scanned.
6	Nominal state. Used only for the binary Bit case.
5	Inhibit. "1": If this channel or bit is bad, assert inhibit control line.
4	Two times. "1": Channel or Bit must be in a new alarm state on two successive scans in order to be recognized as a changed state.
3	Beam only. "1": Only scan this channel or bit on Beam cycles.
2	Not used.
1	Two times counter.
0	Good/bad. "1": bad.

The first 5 bits are control flag bits. The last two are status flag bits.
The inhibit control line is Bit $90. It is assumed memory-mapped. A one means inhibit.
Beam cycles are signaled by the Beam status Bit $9F. A zero means a beam cycle.

Analog alarm scan

For each channel that has the Active bit set, the Beam flag bit is checked. If it is set, but the current cycle is not a beam cycle, then the channel is skipped. The difference between the current reading and the nominal value is compared in absolute value against the tolerance value. If the reading is outside of tolerance, the channel is judged to be bad, and a message is sent if the good/bad bit is zero. (The trip count is also incremented, latching if it reaches 255.) If it is within tolerance, it is judged to be good, and a message is sent if the good/bad bit is a one. The "Two times" bit being set filters this judgment such that the same alarm state must be found on two successive cycles before action is taken. The "two times counter" provides the memory for this determination. The tolerance is a signed value that must be positive; hence, a channel that is 10 volts away from the nominal value is considered to be out-of-tolerance. This restriction could be removed if the tolerance were considered to be unsigned.

Analog alarm message

The format of an analog alarm message that is queued to the network is as follows:

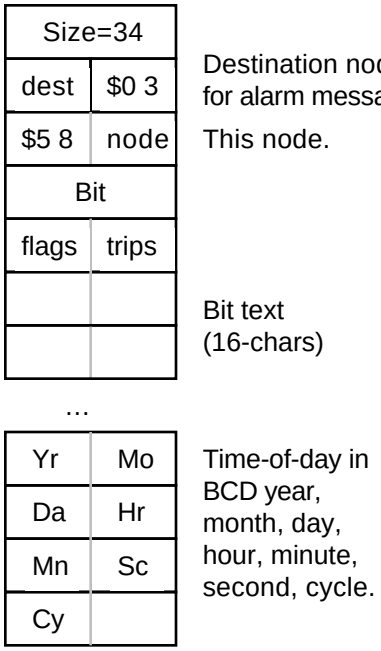
Size=34		
dest	$0 3	Destination node used for alarm messages.
$4 8	node	This node.
Channel		
flags	trips	
Reading		
Setting		
Nominal		
Host Data-base id		Downloaded from host using listype 27.
D	V	Example 6-char analog device name 'DVNAME'
N	A	
M	E	
Yr	Mo	Time-of-day in BCD year, month, day, hour, minute, second, cycle.
Da	Hr	
Mn	Sc	
Cy		

Binary alarm scan

For each Bit that has the Active bit set, the Beam flag bit is checked. If it is set, but the current cycle is not a beam cycle, then the Bit is skipped. If the current Bit reading differs from the Bit's Nominal flag bit state, and the good/bad bit in the alarm flags byte is zero, then send an alarm message to the destination node for alarms. (The trip count is also incremented, latching if it reaches 255.) If the reading matches the Nominal state, and the good/bad bit is a one, then send an alarm message. As in the case of the analog scan, the "Two times" flag bit modifies this determination.

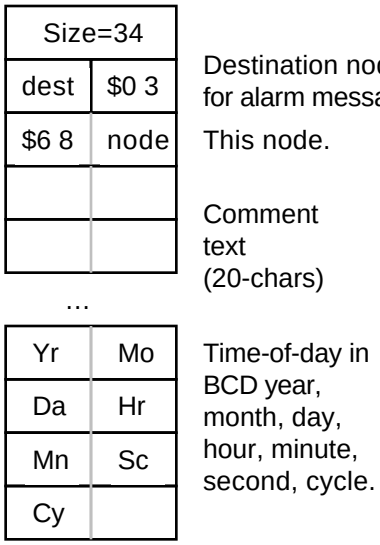
Binary alarm message

The format of an binary alarm message that is queued to the network is as follows:



Comment alarms

A pure text alarm message can also be generated. Currently, it is only used for a message that announces that a VME system has just reset. The Comment routine is included in the Alarms Task module to build such a text message for the network. The format of the network message is as follows:



Timing

The alarm flags byte is scanned sequentially for every channel and bit known to the system. Only channels or bits with the Active bit set are checked for alarm conditions. The three instruction loop that does this is optimized for efficiency. The overhead to scan 256 channels *or* 256 bits *without* the active bit set requires about 250 µ sec on a 133A cpu board. The total Alarm Task time on such an empty system is 0.7 msec. For each active channel or bit that must be scanned, additional time is taken. Rough measurements indicate that it takes 5 µ sec to scan a channel and 4 µ sec to scan a status bit. If a node had a total of 1024 channels and 2048 binary bits, with 200 active channels and 200 active bits, then the Alarms Task would take about 5 msec to make a complete scan. These measurements were made using a 133A cpu board with the 68020 instruction cache enabled.

An improvement that could be made in the alarm scan logic would be to find a way to check only the channels or bits that need to be scanned. A way to handle *binary* scanning more efficiently would be to do logical operations on 16 binary status bits at a time, rather than checking one bit at a time as is done here. But one must weigh the advantage of such changes that would reduce the scan time against the effort required to make the changes.

Analog Control Types

Channel device control

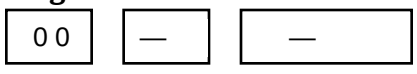
Sep 22, 1989

The Local Station software supports a variety of hardware interfaces. Part of the support is that given to analog channel settings. The information in the analog control field of the analog descriptor for a given channel device describes the specific form of control which is to be used for that channel. This note details the various forms of the analog control field for each type. If the setting action appears successful (no bus error or other errors), the setting word is updated with the data value for the given channel.

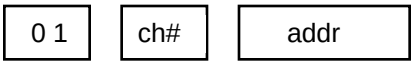
The analog control field consists of 4 bytes. The first byte is the analog control type, a small index value. The meaning of the other 3 bytes depends upon the type byte. The types currently supported are:

00	No analog control (parameter page will not mark it with a "–")
01	Datel Multibus D/A (used in Linac)
02	Motor (setting value is desired reading, relative setting is #steps)
03	Bipolar multiplex D/A (used in Linac)
04	Unipolar multiplex D/A (used in Linac)
05	Memory word (accessed as two bytes)
06	i8253 timer
07	M6840 timer
08	1553 D/A (12-bit)—used in rack monitor
09	Analog Devices RTI-602 D/A board
0A	Memory word (accessed as one word)
0B	Message queue setting to another cpu (co-processor)
0C	Unsigned 12-bit D/A (in short I/O space)
0D	Burr-Brown MPV904 12-bit D/A board
0E	1553 D/A (16-bit)
0F	AMD9513 timer (32-bits from pair of channels)
10	Memory byte (single byte no shift)
11	Memory byte (single byte w/ shift in short I/O space)

Analog control field formats

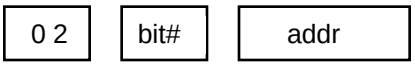


No analog control (parameter page will not mark it with a "–")



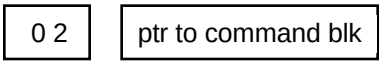
Datel Multibus D/A (used in Linac)

The ch# byte is the board's channel#. The addr word is the board's address (sign-extended). No such hardware in VME systems.



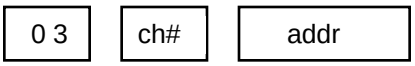
Memory-mapped motor (used in Linac)

The bit# in the range 0–7 is the bit in the byte addressed by the 16-bit address (sign-extended). Motor pulses (~20 μ sec hi-active pulses) are formed at a 150 Hz rate driven by an interrupt from a 68901 timer on the crate utility board.



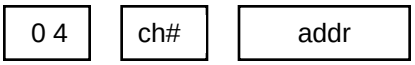
1553-based motor

The ptr must be above $100000 to distinguish this case from the above memory-mapped case. (This is no problem with 1-Mbyte of memory on the cpu board based at $000000.) The command block houses the 1553 command for a single word write of the two's-complement #steps to be issued to the motor. The external hardware is expected to deliver the pulses. The 150 Hz interrupt that is used for the memory-mapped case decrements a counter in order to provide a shadow of the countdown register. This can be used to determine whether the motor is still running, but it assumes that the external hardware's motor pulse rate is also 150 Hz, which may not be the case.



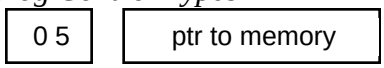
Bipolar multiplex D/A (used in Linac)

The ch# byte includes the chassis# 0–7 and the channel# 0–15 concatenated to form a 7-bit value. The addr is sign-extended to form the address of the D/A.



Unipolar multiplex D/A (used in Linac)

This is the same as type#3, but the setting value is clamped to zero if negative.



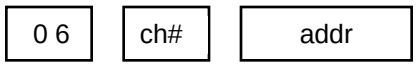
Memory word (accessed as two bytes)

The 24-bit memory address is mapped into a 32-bit address as follows:

If the address is < $F00000, the hi byte of the 32-bit address is $00.

If the address is ≥ $F00000, the hi byte of the 32-bit address is $FF.

This mapping scheme allows entry of short I/O addresses plus all addresses below 15 Mbytes. The data word is written as two consecutive bytes.

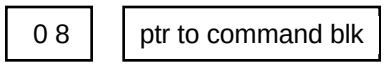


Intel 8253 timer

The ch# is in the range 0–2. The addr is sign-extended to form the address of the 8253 chip. (Not used in VME system)

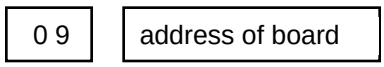
Motorola 6840 timer

The addr ends in 2, 4 or 6 to indicate which of three 16-bit timer channels is to be set. Setting values ≤ 0 are clamped to $0001. (Not used in VME system)



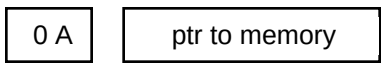
1553 D/A (12-bit)—used in rack monitor

The command block houses the 1553 command for a single word write to the D/A.



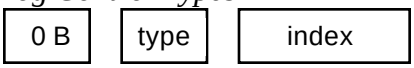
Analog Devices RTI-602 D/A board

The address is mapped into 32-bits via the type#5 scheme. The data value is converted to offset binary and written to the resultant address.



Memory word (accessed as one word)

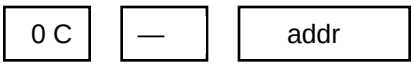
The address is mapped into 32-bits via the type#5 scheme. The data word is written as one 16-bit word. (Some hardware boards require this.)



Message queue setting to another cpu (co-processor)

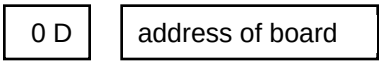
An analog control message is placed into a co-processor message queue. The cpu# is included in bits 6–4 (mask=$70) of the type byte. The message placed into the queue is formatted in this way:

The first word is the size of the message, which in this case is always 8 bytes. The second word is the type byte anded with $8F to remove the cpu#. The third word is an index which may have any value and serves, in conjunction with the type value, to identify what is controlled to the co-processor cpu. Now the message queue to a co-processor is more general than this use for analog control. Other message types may be passed to the cpu by the use of setting commands that use listype #45. The type word used in those messages should not conflict with those used here. (One should not use listype #45 to send the same message as is used for analog control, because the setting word associated with the analog channel will not get updated.) An easy way to insure there is no conflict is to use type word values ≥ $100, since these analog control messages use index words ≤ $8F.



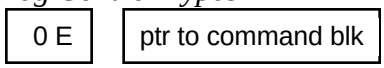
Unsigned 12-bit D/A (in short I/O space)

The addr is assumed to be in short I/O space. The data word is clamped to zero if negative.



Burr-Brown MPV904 12-bit D/A board

The address is mapped into 32-bits by the same scheme used in type #5. The data value is converted to complement offset binary and right-justified to match what the board expects.



1553 D/A (16-bit)

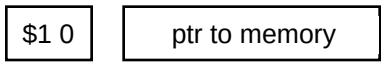
This is the same as type #8, but knob control sensitivity (as used with listype#7) is based upon 16 bits rather than 12.

AMD9513 timer (32-bits from pair of channels)

The addr is taken to be in short I/O space. The code values are of 4 types:

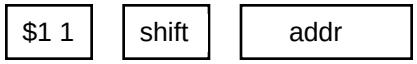
0x Coarse channel
1x Fine channel
2x Clock event selection
3x Clear all events

The x nibble is used to identify the timer channel (0–7 range) that is to be controlled on the clock timer board. More details on this can be found in the document entitled “VME Clock Timer Board.”



Memory byte (single byte no shift)

The lo byte of the data word is written to the given address (mapped to 32 bits using the type #5 scheme).



Memory byte (single byte w/ shift in short I/O space)

The shift count is used to right shift the data word before writing the lo byte to the given address which is assumed to be in short I/O space.

Auto-page on Demand

Do it now!

Jul 6, 1989

Introduction

The Local Station software current supports auto-page application invocation. Associated with each display page is a set of auto-page parameters consisting of a "next" time, a "delta" time and a time-out value. The time parameters are in units of minutes. The "next" time, expressed as yr-mo-da-hr-mn contains the time of the next (automatic) invocation of that page. The "delta" time, expressed as hr-mn, will be added to the current time to produce a new "next" time when the application terminates.

The time-out can optionally force the program to terminate after a period of time up to four minutes. If the time-out period is less than 17 seconds, the current display contents are printed out the serial port upon termination. This allows hard copy records to be produced from programs like the parameter page which have no automatic actions.

When the program does terminate, the page that was "interrupted" is recalled for execution. It won't return to its original state, but it will at least reappear. In the case that another page is also due to be called up, then it will be invoked first before recalling the "interrupted" page.

Demand page invocation

For handling the measurement of the D0 detector argon temperatures and purity monitor, it is desired for the host to be able to request the measurements to be made at times which only the host can determine. The test beam work utilized the afore-mentioned auto-page mechanism to typically make the measurements every ten minutes on a time-of-day basis. For the final experiment, we need to do this upon demand of the host.

An easy approach to providing this is to utilize the present auto-page mechanism to do the hard part. All we need to do is to set in the current time as the "next" time for the given page upon receipt of a special setting from the host. The auto-page mechanism will take over and arrange to run the program as requested. The data of the "setting" can be used to set the time-out value. (See below.) While this may appear to usurp the auto-page ability for the given page, it is probably good that it does. If it didn't, we would need to define what happens when the auto-page time comes up while the page's program is executing under host control. This scheme means that programs which are run upon demand will not be run via periodic auto-page.

Parameter passing

It may be useful to pass parameters to the application that is invoked. When this is done with the usual applications, it is done via the display screen. And often the program retains some of such values in its page-private memory. For each display page, there is a small (currently 120 bytes) private "file" available for keeping such values between invocations of that page. The parameter page, for example, uses this area to keep its current list of channels. In this way, a single copy of the parameter page program can serve many display pages using a different set of channels in each.

This page-private memory can be used to house a record structure that can be used for communications between the host and the application program. The host has access to this same block of memory using listype #33. Parameters can be written there, even before the application is invoked upon demand, Status and progress information that is generated by the application can be monitored by the host. A key could be stored there to verify whether the intended application was running. Error conditions can be reported to the host as well. There is no restriction imposed upon the layout of this structure by the Local Station software; it is organized by arrangement between the application program and the host program.

The host program could check the page's display title to verify whether the page is installed with the expected program. Or, it could merely assume that the page that is used for the program is fixed. It may as well be fixed, as the page number is the ident that is stored in the database anyway; it doesn't need to be known to the host program but only to the database.

Time-out values

The significance of the 2-byte time-out values used in the invocation setting are as follows:

255	Forever
16–254	1–239 seconds (value-15)
1–15	1–15 cycles (.06 – 1 second @ 15Hz)
0	Inactive

Capture Analog Readings

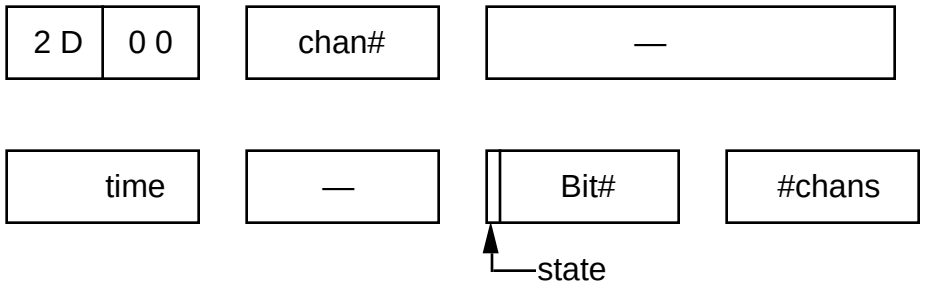
Data Access Table option

Thu, Aug 17, 1995

This option is designed to capture all Linac data readings on a single 15 Hz pulse. One may specify an initial channel number, a number of channels, and an enabling Bit# and state. All analog channel readings in the range indicated are copied into the 8th word of the 8-word ADATA table entries. Listype #28 can be used to request this data using a CHAN ident. The data will remain available until the next time data needs to be captured.

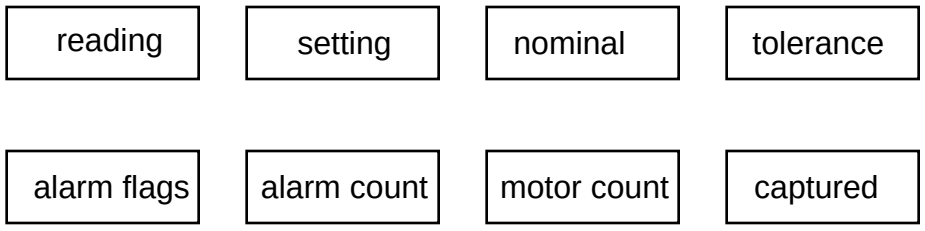
The enabling Bit# might be the beam status bit, for example. This would capture all analog readings on every Linac beam pulse. To make it useful, a host would have to request all this data before the next Linac beam pulse. This would be very difficult to do at 15 Hz. A host program that needed to collect a subset of such data, however, might be able to do it.

RDATA entry format



The time word is a diagnostic *result* that shows the time required to execute this RDATA entry in 0.5 ms units. On a 133A board (68020), the time to do this copy for 1024 channels is 1.0–1.5 ms.

ADATA entry format



Composite Digital Status Words

Acnet adaptations

Nov 13, 1991

Digital data: Local station vs Acnet

The local station software provides access to digital status by Bit#, by Byte#, and by Chan#. The latter uses the associated digital status and control fields in the analog descriptor to provide the parameter page on/off and reset support. Binary alarm scans are made based upon the individual Bit#, so an alarm directs attention to a single bit.

In contrast, Acnet treats a device as including both the analog channel reading and a digital status word reading. An alarm can refer to either the digital status or the analog reading of a single device.

Local station planned changes

Two schemes are planned as additions to the local station. The first is the feature of building words of digital status and assigning them to pseudo analog channel reading words. With a new flag bit in the analog alarm flags word, the analog alarm scan logic is modified to treat the nominal and tolerance words not as numbers, but as a nominal bit pattern and mask. In this way, a set of bits (in the reading word of an analog channel) can produce a single alarm message.

A second scheme is one of enhancing the current associated status and control support to collect up to 8 (or 16) bits of status together. This would permit requesting such status, but it would not include alarm scanning logic on such status. This note focuses on the first scheme.

The problem

The problem at hand is to find some way that combines these efforts into one which can fulfill what Acnet expects and also what the local station expects. These are the requirements:

1. Build status words by device.
2. Report single alarm for the status word.
3. Report analog/digital alarms for single device.
4. Relate status word to analog channel and *vice versa*.
5. Support multiple digital controls for single device.
6. Provide text for each status word bit.
7. Provide state text for each status/control bit.
8. Support named status word if no related analog channel.

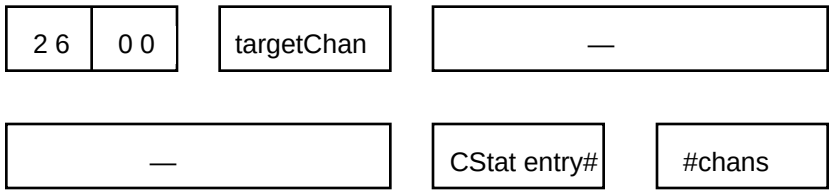
The solution

Suppose that associated status words are constructed each cycle and written into “reading” words of pseudo-channels, according to the first scheme. Also,

suppose that a word in the pseudo-channel's analog descriptor refers to an analog channel that relates to the pseudo-channel. Then the pseudo-channel can be used to construct the EMC for the alarm system.

If these status words are used in the alarm scan, they must be built each cycle, not merely upon a user's request. This implies that it is good to do this efficiently. A table of instructions that describe how to do this efficiently can be interpreted by an offline database uploading utility program in the same way that it is interpreted by processing the data access table in the local station system in order to discover which raw status bits occupy positions in the composite status word.

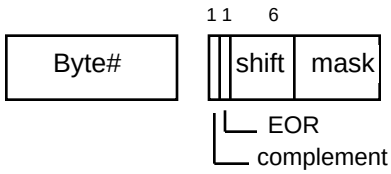
The data access table entry that supports this assembly of status bits is as follows:



The "targetChan" is the first analog channel whose reading word is to be filled with combined status bits. This word will be accessed using the Basic Status property of the central database. The "CStat entry#" is the entry# in the CSTAT system table that contains the list of 4-byte specifications that describe how to assemble the bits together for the first target channel. The system table directory specifies what is the maximum size of each list and hence the offset from the start of one list to the next. The "#chans" word specifies the number of resulting composite status words to be assigned to consecutive channel readings and also the number of lists that are to be interpreted.

If we assume that lists need no more than sixteen 4-byte specifications, as there are only 16 bits in a word, then the stepSize parameter could be 64 bytes. But in practice, it is likely that much fewer than 16 specifications will be needed, as one specification can operate on multiple bits occurring in a raw status byte. One might choose a list size of 32 bytes, for example, to permit up to 8 specs each.

The format of one 4-byte specification is as follows:



The Byte# refers to the raw status byte read by the usual “0405” data access table entry. Raw status byte values are placed into the BBYTE table (#5) by that entry. The mask is applied to the status byte reading as it is positioned in the lo byte of a word with zero in the hi byte. This word is shifted left by the amount of the shift. To shift right, use (16–shift), since it is actually a circular left shift.

The two option bits allow a complement of the status byte before masking and / or the use of exclusive-OR logic rather than the default inclusive-OR for combining the results of multiple 4-byte specs in a list into one composite status word. These two options should *not* be used for filling these composite status words in channel reading words, because they will complicate the job of uploading bit-based text to the central database. They are included for use in building composite status bytes that are placed into table #5 by using $2605 in the data access table entry, rather than $2600. (Table #0 is the analog data table.) In that case, new status bit text can be entered, and such new bits can be further combined into channel reading words as above. The whole matter of these two option bits can be considered an advanced topic that is unneeded for awhile.

In forming these 4-byte specs, the Byte# word must have a value in a valid range. The value zero is not considered valid, even though Byte #0 exists. The reason is that it is likely that many spaces reserved for specs will be unused and left zero.

It will take some care in devising the appropriate specs, but that is the nature of what is needed for doing the job of constructing these composite status words. Interpretation of the specs is expected to be quite efficient, which is desirable as it has to be done at 15 Hz.

Note that a single bit can be used as a composite status word in a trivial way using only one spec. This can give that bit an Acnet-style name. Raw status bits in a local station do not currently have names *a priori*.

To relate a composite status word “channel” to a real analog name, one could use the family word in the analog descriptor for that purpose. Alternatively, special (but similar) names can be defined for these composite status words.

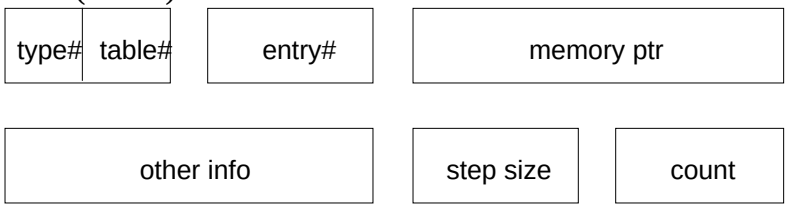
Data Access Table Formats

Data pool preparation

Thu, Jul 25, 1996

Introduction

The Data Access Table is used to specify what happens within a station every cycle to prepare the data pool. Included in this is a mechanism for executing all enabled closed loops and server code. At the start of a periodic cycle, the Update Task is executed. As part of its work, entries in the DAT are processed from beginning to end. Each entry is an "instruction" to be interpreted before moving on to the next entry / instruction. The generic pattern of such entries is as follows, shown as 8 (16-bit) words:

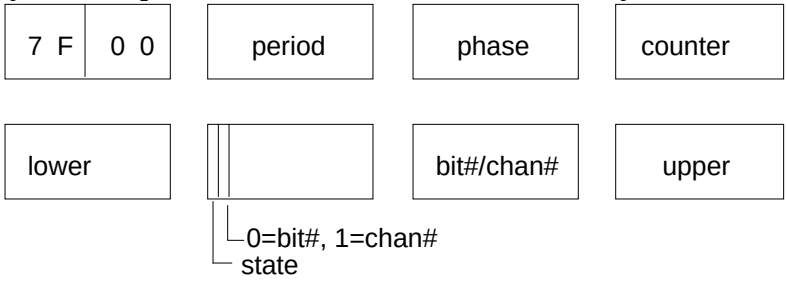


The hi byte of the first word specifies the entry type#. The lo byte of the first word is a system table# if it is in the range 00–1F. The meanings of all other fields depend upon the entry type#. The above layout of these entries is only an example. Normal entry type#s are positive integers in the range 01–7F. Entry type#s of 00 or in the range 80–FF are invalid and are ignored during DAT processing. Entry type 7F is the condition entry.

The table# is normally either 00 (for the channel entries in the Analog Data Table) or 05 (for the byte entries in the Binary Data Table). The entry# is the entry of the first data word targeted. The data words are copied into successive entries of the destination table when a count is specified. Most often, the entry# is a channel#. Some types use an address pointer which provides a hardware board address or other memory pointer. The 7th word is often a step size when a memory ptr is used. The count word is usually a loop count of the number of consecutive channel readings (destination table entries) to be filled.

Condition entry

Entry type 7F is a special entry that is used to enable / disable an internal flag that determines whether a non-7F entry is processed or skipped. At the beginning of DAT processing, this internal flag is initialized to disabled, so that a 7F entry must occur before any non-7F entries will be interpreted for processing. The only result of processing a 7F entry, which by definition is never disabled, is to enable or disable the internal flag that determines whether a subsequent non-7F entry will be processed. The format of a 7F entry is as follows:



At its simplest, this entry merely specifies the period word in cycles. For

processing every cycle, use 1. Such is typically the first entry in a DAT. If processing is desired every other cycle, use period=2. The other options allow for enabling the internal flag based upon the state of a bit, or the value of an analog channel being inside or outside a specified range. For more information, see the document RDATA Periodicity.

Overview

As an overview of the entry types available for use as DAT instructions, here is a list by type#:

01	Multiplexed A/D used by Linac
02	(n.u.)
03	Read memory words by bytes
04	Read binary bytes via address list
05	Shift data words
06	Adjust nonlinear RF diode readings
07	Zero-data pedestal adjustment
08	Compute ratio
09	Compute product
0A	Compute sum
0B	Compute difference
0C	Process 1553 command list
0D	Auto-setting from memory
0E	Wait, post-process 1553 data
0F	Average sequence of readings
10	(n.u.)
11	Analog Devices A/D board
12	Sample datapool
13	Read memory words by words
14	High Voltage Digitizer (obs)
15	Beam status counter
16	Capture data on selected cycles (obs)
17	Timer channel clock event counts (obs)
18	Timer channel clock events (obs)
19	AMD9513 timer delays (obs)
1A	Read single bytes of memory
1B	Read words—mask,shift,BCD options
1C	Read clock events from clock board (obs)
1D	Invoke local applications
1E	Insert data into memory words
1F	De-multiplex data words
20	Send data request to SRMs
21	Wait for SRM data reply
22	Map SRM data into data pool
23	(n.u.)
24	Compute counter differences
25	Copy setting word
26	Assemble combined status words
27	Copy memory into data stream
28	Copy from IRM A/D circular buffer
29	De-multiplex binary data bytes
2A	Copy words memory-memory
2B	Copy bytes memory-memory
2C	Copy FIFO to memory
2D	Save all readings in present cycle

Some entry types access data from various hardware interfaces. Some modify data already collected in various ways. In particular, entry 1D allows for processing all local applications, some of which may generate output data for inclusion in the data pool. Much of the particularization, or configuration, of a station stems from the design of its data access table entries.

Editing DAT entries

Armed with the detailed specifications, one can enter DAT entries using a memory dump page. If this is done, one should take care, as the DAT is scanned every cycle, so it is "live." As changes made in this way are usually made one word at a time, it may be wise to disable an entry while it is being modified, say by setting the $80 bit in the type# byte. Remove this sign bit when the rest of the entry is ready. This form of raw editing of the DAT is not for the squeamish.

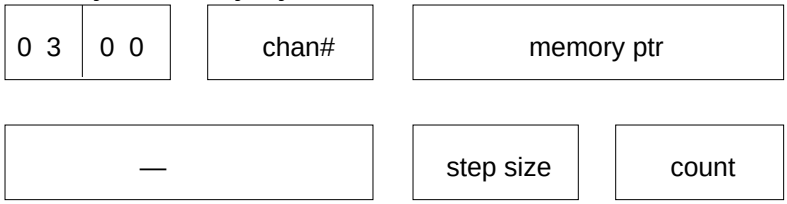
Another means of editing the DAT is provided by a Unix tool called xxxxx. It operates by reading up the entire DAT and producing a text file version of it, which is then edited and downloaded all at once. See document xxxxxx.

DAT entry formats

A brief description follows for each DAT entry type, grouped into related types. In some cases, additional details can be found in other related documents.

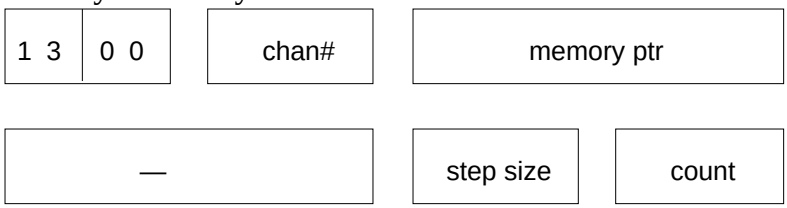
Accessing memory data

Read memory words by bytes



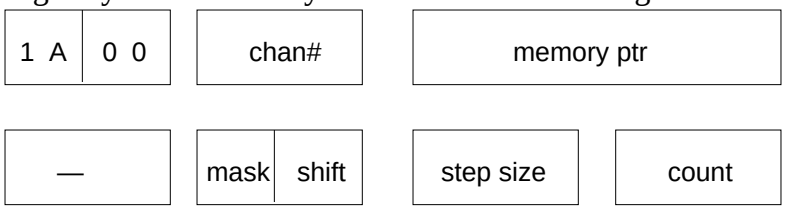
Words are copied (accessed by bytes) starting at the memory address given, advancing by the step size for each destination table entry (*chan#*). If the *memory ptr* refers to data stored in consecutive words, the *step size* would be 2.

Read memory words by words



Words are copied (accessed by words) starting at the memory address given, advancing by the step size for each destination table entry (*chan#*). If the *memory ptr* refers to data stored in consecutive words, the *step size* would be 2.

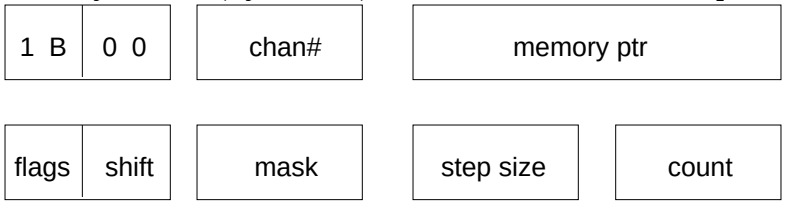
Read single bytes of memory and convert to reading words.



For the range of selected channels, read a byte from memory, apply an optional mask, shift an optional amount and use the resulting 16-bit word as a reading. If the *mask* is zero, no masking will be applied. If the *shift* count is negative, a right shift of (*–shift*) bits is indicated, starting from the data byte positioned in the hi

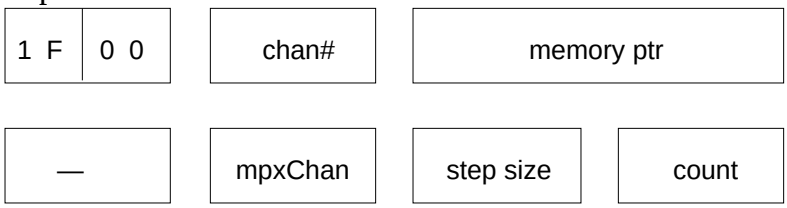
byte of the word and zero in the lo byte. If the *shift* count is positive, a left shift of that many bits is indicated, starting from the byte positioned in the lo byte of the word and zero in the hi byte. The *step size* is used to advance the *memory ptr* when more than one byte is accessed (*count* > 1).

Read memory words (by words) with mask, shift, BCD options



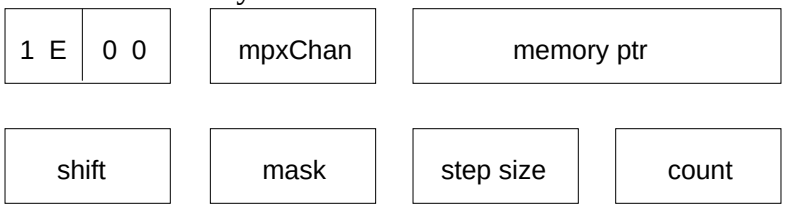
Copy words of memory (accessed by 16-bit read cycles) into consecutive channel readings. Apply optional *mask* (0 treated as $FFFF), optional *shift* (positive=left, negative=right, zero=none), and optional BCD-to-binary conversion (*flags*=$80 to enable conversion). The *step size* advances *memory ptr* for *count* words.

De-multiplex data words



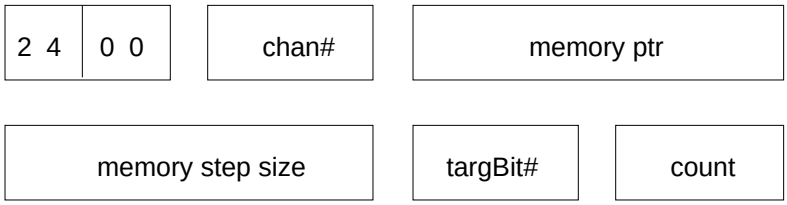
De-multiplex words of memory data according to the value of the *mpxChan*. The value of *mpxChan*, for example, may range from 0–F on successive cycles. Data from memory (count words using step size) is copied into the readings of channels numbered from (*chan#*mpxValue*) to (*chan#*mpxValue* + *count* – 1). This is useful when the hardware interface furnishes multiplexed data according to a value supplied on digital control lines. Type 1E may be used to place the proper value on the control lines.

Insert data into memory words



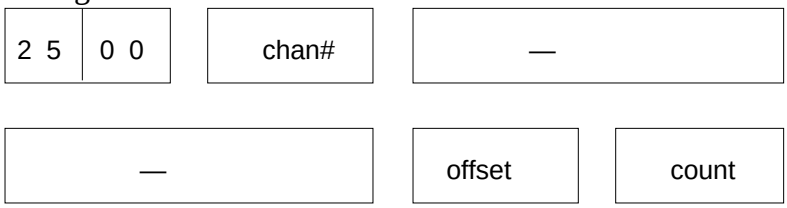
Sample masked value from *mpxChan* and insert into memory words. The reading of *mpxChan* is masked by *mask* and left-shifted by *shift* bits (rotated as a 16-bit word, so use 16–n to shift right) and inserted into the target memory word(s). The bits outside the mask in the target word(s) are not modified.

Compute counter differences



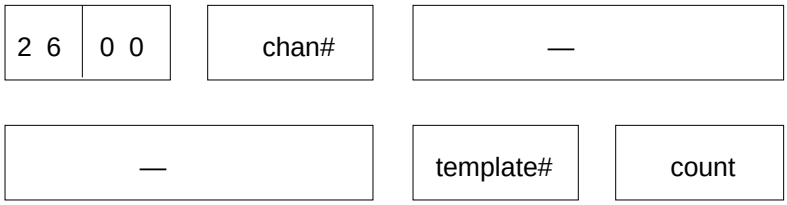
Monitors memory word counter differences. This can be used to monitor whether an associated cpu in the same VME crate is still working by watching a counter word at *memory ptr* that the cpu increments regularly. Optionally, if *targBit#* is nonzero, a bit# can be set if the difference from last time is nonzero, and cleared if the difference is zero. When *count* > 1, additional memory counter addresses are obtained from using memory *step size*, and successive bit#s are used when *targBit#* is nonzero. The memory of what the word read last time is retained in the setting word of the associated target difference *chan#*, so such channels cannot be settable.

Copy setting word



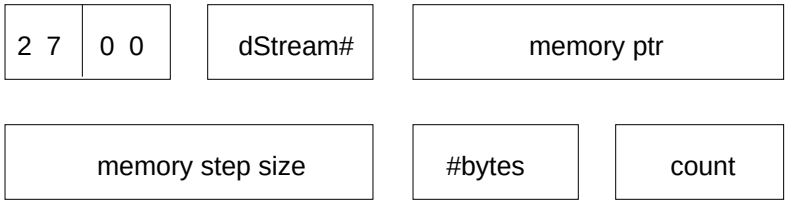
Copy setting (or other) field values into reading fields of successive analog channels. The *offset* value is the offset to the required field in the ADATA table entry relative to the reading field. Use *offset* = 2 for setting field values.

Assemble combined status words

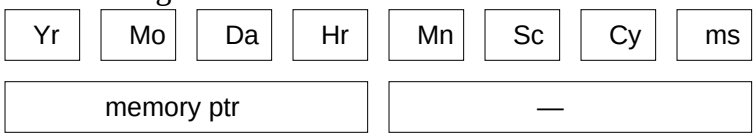


Assemble words of status from collections of bits found in the BBYTE table, using templates found in the CSTAT table #24. Each status word is built from a template found in this table. The reading of *chan#* is built from *template#*, and the process is repeated for successive channels and templates according to *count*. The template is an entry from the CSTAT table, each of which consists of up to 8 specifications of 4 bytes each. Each specification is a Byte# word, followed by a shift count byte and a mask byte. See document Composite Digital Status for more details.

Copy memory blocks into data stream



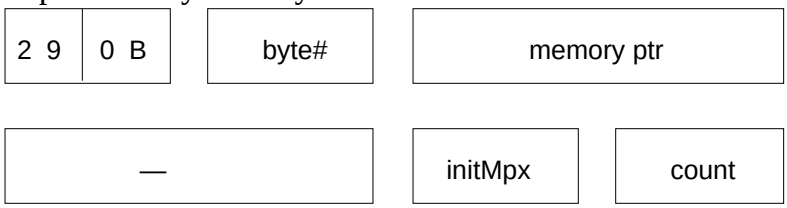
Copy a block of memory of size *#bytes* from *memory ptr* into a data stream with index *dStream#*. The beginning of the record written contains a 16-byte header with the following format:



This header includes the time-of-day the record was written, followed by the memory address from which the block was copied. If *count* > 1, then multiple blocks of memory can be so captured, each including a header. In this case, subsequent block source memory addresses are derived using the memory *step size*. The data stream should be defined in the DSTRM table to have records whose size reflects both the header size and the data block size. For example, if 1024-byte blocks of memory were to be captured into a data stream, the data stream record size as defined would be $410. The time-of-day format is BCD, except for the *ms* byte that holds the residual milliseconds of the present cycle. The *Cy* byte ranges from $00–14, indicating the present 15Hz cycle.

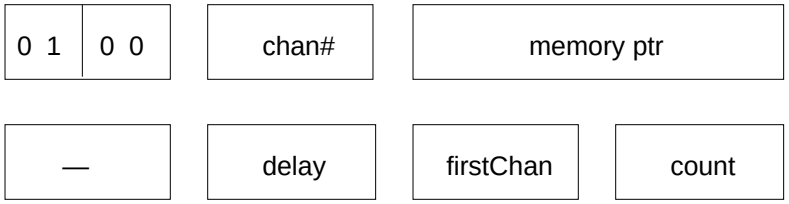
Access to specific hardware interfaces

De-multiplex binary data bytes



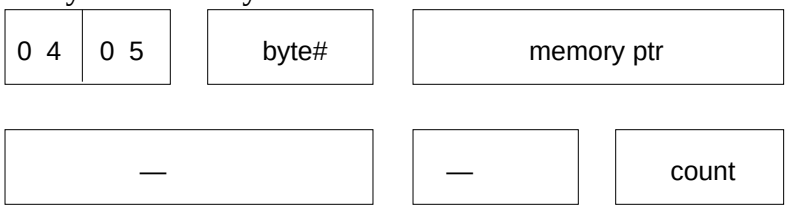
This entry assumes a simple hardware interface for multiplexing binary data bytes. The *memory ptr* is the address of a multiplexed data byte; *memory ptr+1* is the address of the multiplex select byte. The *initMpx* is the initial value of the multiplex select byte; subsequent values are merely incremented from that. The *byte#* is the initial entry used in the BADDR table for obtaining the target addresses for the data byte read from the multiplexed data byte. This simple multiplexing scheme may be used to bring in many digital data bytes using only two bytes of I/O interface. (Each IRM, for example, includes an interface to eight bytes of digital I/O. In the Fermilab Booster HLRF system, two of these bytes are used in this scheme to bring in 16 bytes of multiplexed digital status.)

Multiplexed A/D used in Fermilab Linac



This entry accesses A/D data as interfaced via the original multiplexed A/D system in use at the Fermilab Linac. The SRMs have since been used to read this data, so this entry is no longer needed.

Read binary raw data bytes

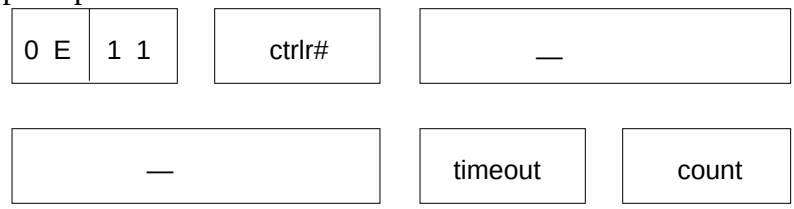


An array of byte addresses (usually the BADDR table) specified by memory ptr contains pointers to consecutive bytes of binary status data. They are treated as memory-mapped data bytes unless the high byte of the address found is $80, 81, or 82, which carry special significance. [If the high byte is 80, the entry is assumed to be a pointer to a 1553 data byte in a 1553 command block on the 1553 controller board's memory. If the high byte is 81, the next byte is an SRM address, and the last two bytes is the control value needed to be sent to the SRM for setting the byte. If the high byte is 82, the next three bytes specify parameters needed for PLCQ message queue processing.] In other cases, the 4-byte entry is a memory address that is accessed to obtain the data byte stored for the byte# given (in the BBYTE table). The number of successive entries filled in BBYTE is given by count.

Process 1553 command list

The pointer is used to process a sequence of 1553 command blocks, each of which executes one 1553 command. Each command may result in up to 32 data words transferred. The *count* word in this case indicates the number of command blocks to be processed. For each word read by a command block, a new reading is stored in consecutive channels. In the binary data case, with the table#=5, each word read produces two consecutive bytes of binary status readings. Separate queues are maintained of commands awaiting execution by multiple 1553 controllers. The interrupt following completion of one command passes the next command, if any, to the controller.

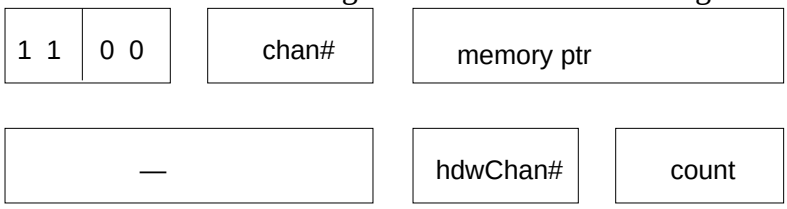
Wait, post-process 1553 data



Wait for the 1553 interrupt activity to complete. Systems which do 1553 I/O with interrupts allow overlapping of multiple 1553 controller activity during DAT processing. This entry must be used to wait for all the readings which have been queued up for interrupt-driven acquisition to finish. This post-processing of 1553 data collection also copies the data into the readings field of the ADATA table, so that this entry must be included. The *timeout* word specifies the time within the cycle (in 0.5 ms units) after which to give up awaiting all controllers in the range specified by *ctrlr#* and *count* and continue DAT processing of any remaining entries.

Analog Devices A/D board

This is the driver for a VME digitizer board from Analog Devices.



Here, *memory ptr* is the base address of the board. and *hdwChan#* is the initial hardware channel select.

Send data request to SRMs

2 0	0 0	—	—
SRMnode#	reqType	#bytes	—

Smart Rack Monitors (SRMs) are used in the Fermilab Linac. As many as 5 SRMs are connected via ARCnet to a single VME station. This entry sends out a request message for data to be returned from an SRM. The *SRMnode#* is usually $7A00, specifying broadcast to all SRMs. The *reqType* is $2201 in the case of requesting the SRM to read and return all its normal cycle data. The *#bytes* specifies the maximum size of the return data buffer. This DAT entry does not wait for the response from the SRMs. That function is specified using the next entry. For more details, see the document SRM Message Protocols.

Wait for SRM data reply

2 1	0 0	—	—
SRMnode#	deadLine	—	—

Await responses from a specific SRM. The *deadline* word specifies the maximum time within the current cycle to wait, in 0.5 ms units. In response to a broadcast request, the order of SRM responses is not determined. But the system's SRM support knows which have responded since the request was sent. It is necessary to place this DAT entry before any $22 entries that refer to the same SRM node#. See the document SRM Message Protocols for more details.

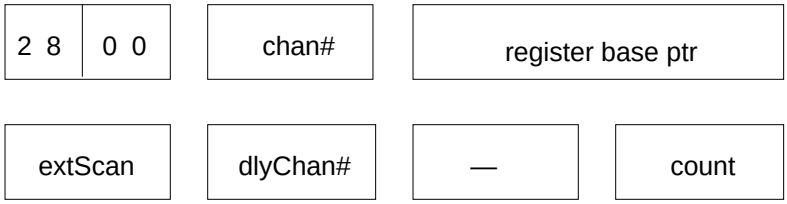
Map SRM data into data pool

2 2	0 0	chan#	—
SRMnode#	offset	table#	offset	count
2 2	0 5	byte#	—
SRMnode#	offset	table#	offset	count

These entries process the already-received response data from an SRM and copy it

selectively into the data pool. The *table#/offset* word identifies the SRM data segment of the response buffer that is to be mapped to the channel or byte data. See the document SRM Message Protocols for more details.

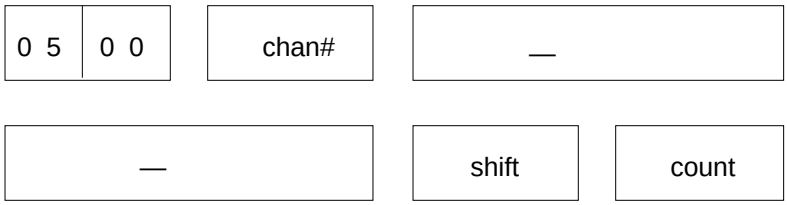
Copy from IRM A/D circular buffer



The IRM analog IndustryPack board maintains a 64K-byte memory that is updated by the hardware with 64 channels of analog input digitized and stored every millisecond. There is room for 512 samples of such data, covering about 0.5 second of time. This entry usually copies the most recently-digitized set of 64 readings into the data pool. If *dlyChan#* is nonzero, it backs up to a time within the current cycle given by the reading of the indicated channel. If *extScan* has the least bit set, it causes the A/D interface to use an external trigger for its digitaizer scan. This option is needed for the PET project, where the scan rate is 360Hz rather than 1000Hz. The *register base ptr* refers to the analog IP board's register address. It is usually FFF58300.

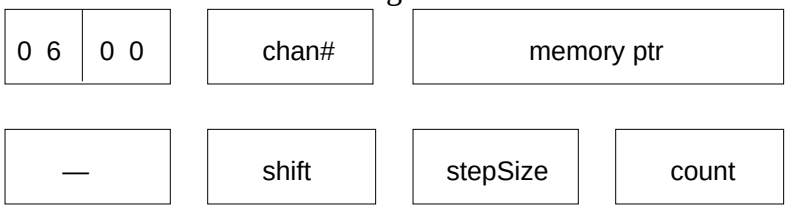
Modify/compute data already acquired

Shift data words



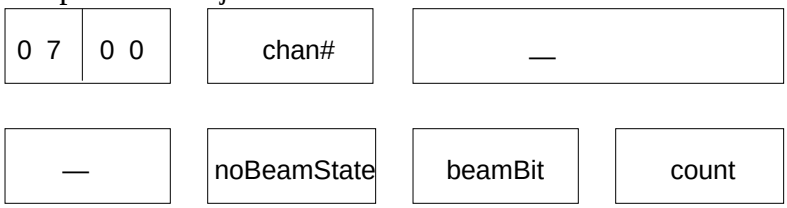
Shift reading fields of a sequence of channels. If *shift* is negative, right shift reading word with sign extension. If *shift* is positive, left shift reading word with zero fill. This has been used to adjust 12-bit A/D readings based on a 2.5 volt scale so that they appear to come from a 14-bit A/D with a 10 volt scale. This is a replace operation.

Adjust nonlinear RF diode readings



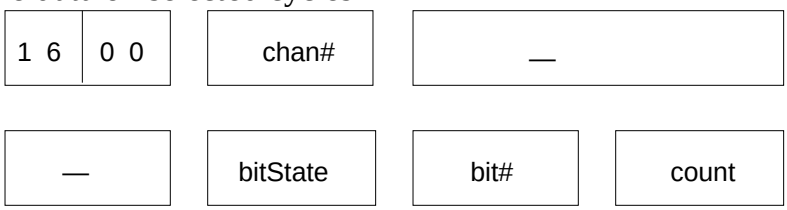
Certain RF amplitude and power readings encountered in the Fermilab Linac system were measured by detector diodes and therefore have nonlinear characteristics. This entry linearizes the readings so they can be linearly scaled in higher level programs the same as any other analog channel readings. Channels in the indicated range from *chan#* to *chan#+count-1* were linearized according to one of two formulae if specified by flags in the "conversion flags" field of the analog descriptor. Flag bit#3 specifies that linearization is to be performed; bit#0 specifies either gradient (0) or power (1) linearization algorithm. If *stepSize* is nonzero, the nonlinear data is taken from memory beginning at memory ptr rather than from the present reading field of the target channel. The *shift* word specifies a shift applied to the raw data word before linearization. See document xxxxx for more details on the linearization algorithms used.

Zero-data pedestal adjustment



Perform automatic pedestal subtraction for selected channels in the target range specified by chan# and count. If *beamBit* is nonzero, it is an optional beam status bit whose no-beam state is given by the sign bit (bit#15) of *noBeamState*.. If *beamBit* is zero, the default beam status Bit# (009F) and no-beam state ($8000) will be used. Each channel to be so treated must be indicated by the appropriate flag bit set (bit#2) in the "conversion flags" byte in the analog descriptor. The result of this logic is that readings read exactly zero, by definition, for cycles in which there is no beam. The pedestal value is kept in the setting word of each channel so treated, so the channel cannot be settable. It may, however, have motor control, since motor-controlled channels do not have setting values. In order for the *beamBit* status to be valid, this entry should occur in the DAT *after* the type 04 entry that updates the BBYTE table with binary status bytes.

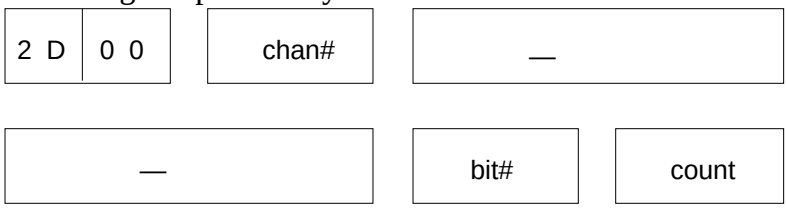
Capture data on selected cycles



Scan the readings of a sequence of channels and capture the reading values for each channel in the range that is marked to need this treatment in its analog descriptor via bit#1 of the "conversion flags" byte. The capture is done on cycles when the status *bit#* matches the bit state given (in the sign bit of *bitState*); otherwise, the captured reading is copied over the current reading, thus

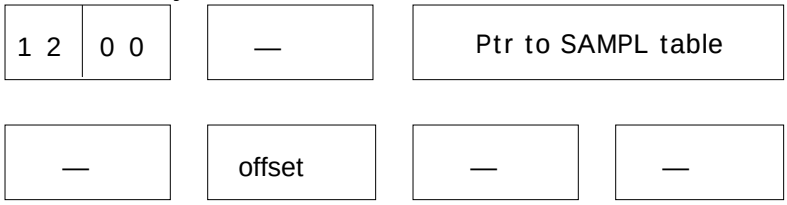
preserving the reading that had been captured before. In systems with channels whose data is valid only during some selected cycles, this entry allows preserving only valid data readings in the local data pool. When a host computer requests data from such channels, it will find only the most recent valid readings there. If it is necessary to also have the current readings, another channel with a copy of the same channel's reading could be set up normally. The captured data values are written into the 7th word of an ADATA entry. Note that motor control cannot be used for such channels, since that same word is used as a motor countdown word in that case.

Save all readings in present cycle



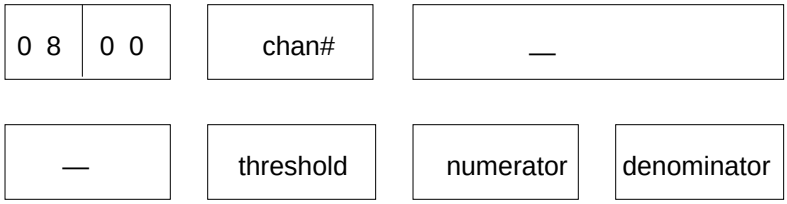
For the range of selected channels specified by *chan#* and *count*, capture the present reading fields into the 8th word of the ADATA entries. The *bit#* word specifies in the lo 15 bits the status bit# that determines whether this capture operation is performed. The state is indicated in the ms bit of this same word. After capture, a host may want to retrieve these values using the listype# defined for accessing the 8th word of an ADATA entry—before the next occurrence of the same status bit state.

Sample data facility



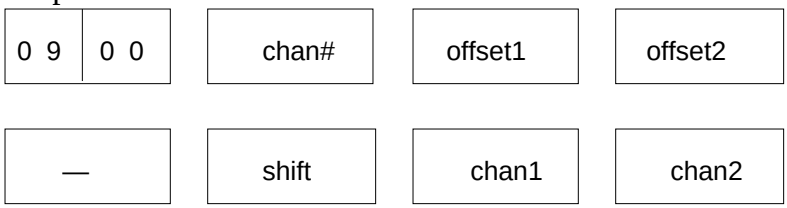
From parameters stored in the SAMPL table, copy a set of channel readings from the local station to memory (especially on the Vertical Interconnect). A table is built containing pairs of words, each of which has a channel# word followed by the data value word. Additional details on this are found in the document Sample Data Facility for VME Stations. This was used by D0 in the "early days."

Compute ratio



Compute ratio between two analog channel readings, where *numerator* and *denominator* are channel#s, and *theshold* is the value of the denominator channel such that, if the absolute value of the denominator reading is below it, the result *chan#* reading will be zero; otherwise the result will be numerator / denominator expressed in volts; i.e., if the readings are equal, the result will be one volt, or $0CCC. The standard full scale range is 10 volts. If an overflow results, use +/– full scale, as appropriate.

Compute product

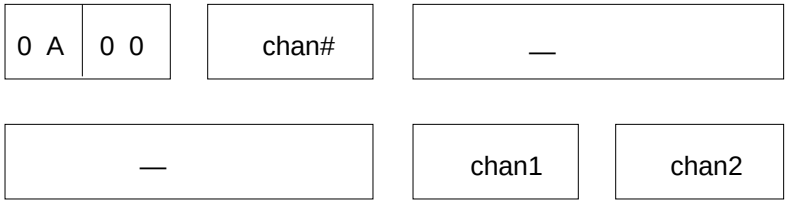


Compute product of two channels *chan1* and *chan2*, and scale by shift. The complete formula used is:

(*chan1*.reading–*offset1*)*(*chan2*.reading–*offset2*)*2^*shift*

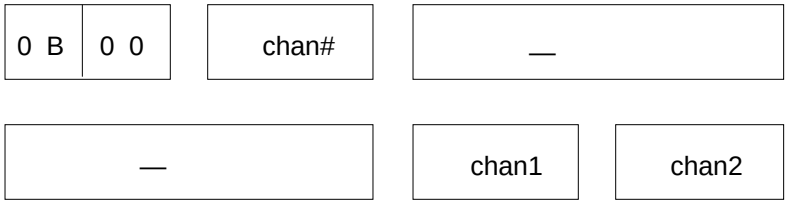
Note that the values of the two offsets are constants, not channel#s.

Compute sum



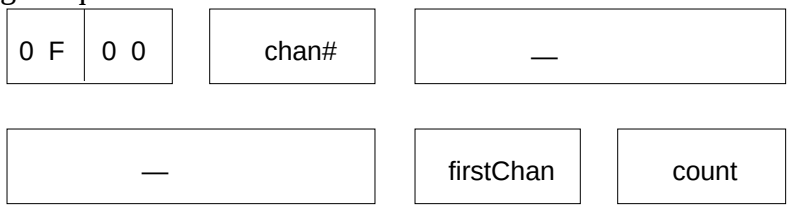
Compute sum of two channels *chan1* + *chan2*. Divide result by 2 in order to prevent overflow. As an example, if the full scales of two readings were both 100.0 amps, then to derive a *chan#* reading that is the sum of the two, the full scale of the result channel should be 200.0 amps.

Compute difference



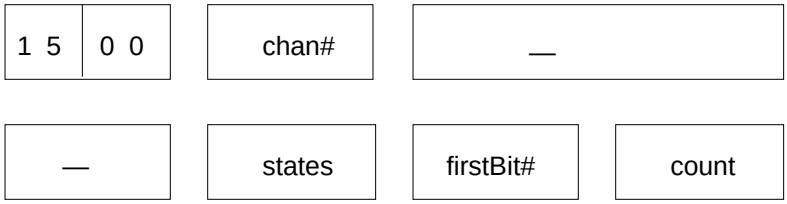
Compute difference of two channels *chan1 – chan2*. Divide result by 2 in order to prevent overflow. As an example, if the full scales of two readings were both 100.0 amps, then to derive a *chan#* reading that is the difference of the two, the full scale of the result channel should be 200.0 amps.

Average sequence of channels



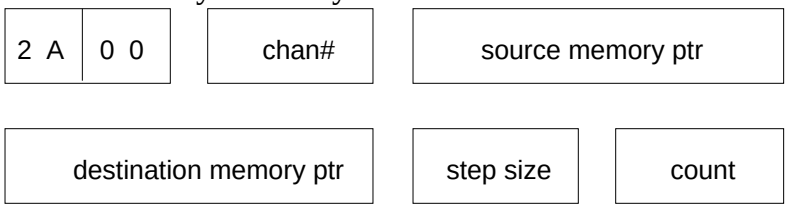
Average the sequence of channel readings from *firstChan* to *firstChan+count–1* and place the result in the reading field of*chan#*.

Beam status counter



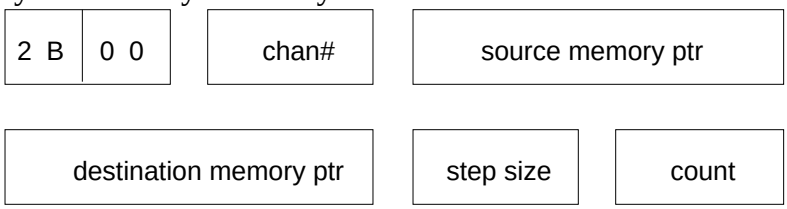
Produce counter readings in target channels by sampling bit# states of sequential status bit#s. The first bit state, to be compared with the *firstBit#* status, is in the sign bit of the states word. Successive status bit#s are compared to successively lower-numbered bits in the *states* word. This naturally limits *count* to 16. When a status bit matches the indicated state, the counter is cleared; when it differs, the counter is incremented. One use of this would be to build a channel whose reading is a counter that measures the #cycles since the last beam cycle. This feature is described further in the document Monitoring Counters.

Copy words memory–memory



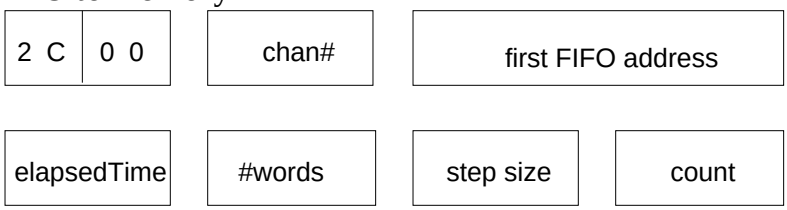
Copy *count* memory words from *source memory ptr* to *destination memory ptr*. The *step size* is used to advance the *source memory ptr*. If the source words are consecutive, then *step size* = 2.

Copy bytes memory–memory



Copy *count* memory bytes from *source memory ptr* to *destination memory ptr*. The *step size* is used to advance the *source memory ptr*. If the source bytes are consecutive, then *step size* = 1.

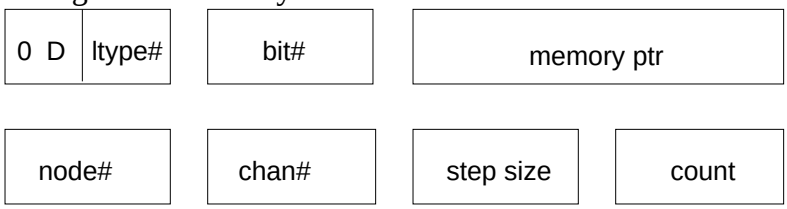
Copy FIFO to memory



Copy FIFOs contents into memory. The *count* word specifies how many FIFOs to read out. The *#words* specifies how many words to read out from each FIFO. The *step size* is used to advance to th next FIFO address. The destination address is found from the analog control field of each *chan#* in sequence. Such channels may be called waveform channels. (This method was used for the first version of the swift digitizer IP module. A later version added memory to the board so that readout of the FIFOs by the *CPU* was no longer necessary.)

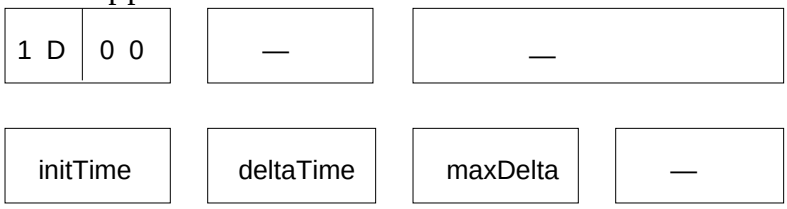
Miscellaneous

Auto-setting from memory

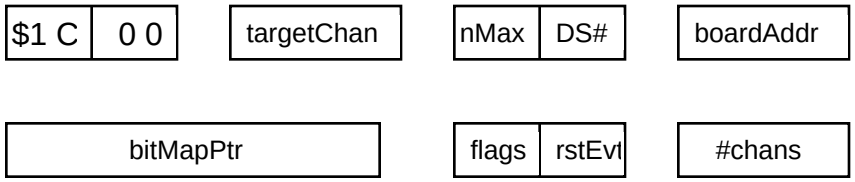


Memory words are copied as setting data, where the channels to be set are consecutive starting at *chan#* in *node#*. This is a way to turn memory data words into readings in another station, although it can, of course, also reference channels in the local station by using the local node#. If the *bit#* (with state value in the sign bit) is nonzero, it conditions the setting action upon the state of the indicated status bit.

Invoke local applications



Scan all entries of LATBL (local application table) in sequence. For each entry that is enabled, call the named local application, including the appropriate value for the call type: initialize, terminate, or cycle. In this way, every enabled local application is invoked every cycle, giving it a chance to perform whatever it needs to do on that cycle. Every LA instance must specify an enable Bit# as the first parameter in its LATBL entry. When the bit is set, the instance is enabled. The three time word indicated above are diagnostics, all in 0.5 ms units. The *initTime* word is the time of starting this DAT entry within the current cycle. The *deltaTime* word is the total cpu time used by all the enabled LA's this cycle. The *maxDelta* word is the maximum value of *deltaTime* ever. Since this DAT entry executes closed loops, it is usually one of the last entries in the DAT, so that it has access to the latest values in the data pool.



Three jobs can be done by this entry with the clock event data. Clock event data is written to the Clock Event Queue (as a data stream). A range of analog channel readings is updated with the most recent time-stamps for the corresponding events (starting with event $00). A bit-map is updated with bits set to indicate the occurrence of the same range of clock events.

The `targetChan` is the base channel number of the range used to hold the time-stamp data, likely with the option enabled to store these time-stamp data relative to the time of the cycle reset event. The `nMax` byte is the maximum number of times the hardware fifo is read while processing this entry. The `DS#` byte is the data stream index used to identify the Clock Event Queue data stream according to the `DSTRM` table. The `boardAddr` is a word that gives the Clock Decoder boards fifo address in VME Short I/O space.

The `bitMapPtr` is a pointer to the base of the bit-map array which contains bits set to 1 whenever the corresponding event occurs. The bit-map may optionally be cleared at cycle reset time. The `flags` byte includes the following options:

bit#	option
7	1= enable cycle reset event logic
6	1= enable bit-map updating
5	1= enable bit-map clear upon cycle reset
4	1= longword bit-map array, 0= byte bit-map array
3	spare
2	spare
1	spare
0	spare

The cycle reset event# is specified by the next byte, used only if the cycle reset event logic is enabled. The `nChans` word is the number of channels starting at `targetChan` that are used to record the corresponding time-stamps for events in the range 00n, where n= (`nChans-1`). It is also used to give the range of events that are recorded in the bit-map table, if the bit-map is enabled.

D0 Alarms

Local Station Implementation

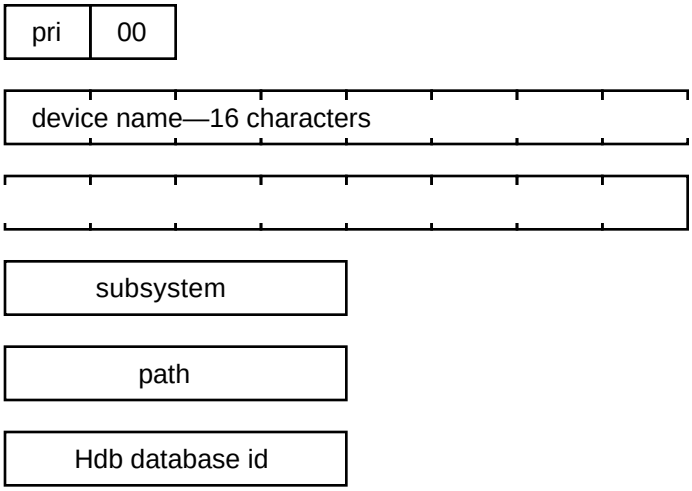
Feb 5, 1990

New alarm message protocols are required for use by D0 in order to conform with the "D0 CDAQ Network Data Transmission Protocol" document by Alan Jonckheere. Alarms messages are assumed to use Acnet header-style messages designed by Charlie Briegel on the token ring network. This document describes the implementation of D0 alarms in the Local Station software.

Device Information Blocks

Alarm parameters must be downloaded from the D0 Host to assist the Local Station in building the alarm messages that are required. This allows the Host alarm processing to avoid accesses to its own Rdb database as much as possible in the interest of execution efficiency.

There are three types of alarm messages—analog, binary and comment. For each there is a Device Information Block that must be downloaded in the following format:



Three new system tables are used to contain these parameters for the three types of alarms. The tables numbered #21, #22 and #23 are named `AADIB`, `BADIB` and `CADIB`, for Analog Alarm Device Information Block, Binary Alarm DIB, and Comment Alarm DIB. Each is allocated 32 bytes/entry to include the 30 bytes above *preceded* by a word used to keep a date-of-last-change for that table entry. (By preceding the DIB with this word rather than following it, the name is kept quad-aligned in order to optimize memory access for name searches.) The number of entries for each table is the same as the number of analog Channels, the number of binary Bits, and the number of Comments, respectively. New listypes support access to these tables both for downloading and for reading access.

Alarm Control Blocks

These structures are also downloaded when alarm flags and/or associated parameters such as nominal and tolerance values are set. In this case, there already exists an equivalent data structure in the current system. To support a faç ade of the new structures, additional read and set type routines are used. These routines allow access to the different structures used internally through an interface that D0 specified. If the underlying internal structures change in the future, then these routines can be modified as needed without requiring changes at the Host level. (This approach might be compared to the "methods" used in object-oriented languages.)

Local alarms display

To support display of local alarm messages, the Alarms Task passes the current alarm message with the "used" bit set through OUTPQ for later processing by QMonitor *in addition* to the D0-specific alarm message that is queued to the network. When the Alarms Task receives a message from another station (source lan-node not equal to its own) directed to the task named ALRM, it allocates a memory block to contain the alarm message but builds it in the *classic* form. In this way, the classic support for alarm message encoding into ascii (for display on the bottom line of the local console or via the local serial port) still works.

Comment alarm data

A new system table #10 named CDATA provides space for data associated with comment alarms. This data includes a comment alarm flags word, an alarm count, and the text to be used with the comment alarm message.

LTT (Listype Table) changes

listype#		ident	read#	set#	#bytes	tbl#	offset	purpose
56	16	1	1	2	10	0		CDATA access
57	1	16	18	10	0	0		D0 Analog alarm ctrl
58	2	17	19	2	2	0		D0 Binary alarm ctrl
59	16	18	20	32	10	0		D0 Comment alarm ctrl
60	17	0	21	2	0	0		General resets
61	1	1	1	30	21	2		AADIB access
62	1	1	1	16	21	4		Analog name
63	1	1	0	0	21	0		AADIB date of last change
64	2	1	1	30	22	2		BADIB access
65	2	1	1	16	22	4		Binary name
66	2	1	0	0	22	0		BADIB date of last change
67	16	1	1	30	23	2		CADIB access
68	16	1	1	16	23	4		Comment name
69	16	1	0	0	23	0		CADIB date of last change

Ident type#s

16 Comment index
17 General reset index

Read type#s

16 Analog alarm flags, nominal, tolerance
17 Binary alarm flags
18 Comment alarm flags

Set type#s

18 Analog alarm flags, nominal, tolerance
19 Binary alarm flags
20 Comment alarm flags
21 General resets

New table#s

10 CDATA Comment Data Table
21 AADIB Analog DIB
22 BADIB Binary DIB
23 CADIB Comment DIB

Family Alarm Messages

Displayed alarm message reduction

Mar 7, 1992

Introduction

The local station alarm scan logic was designed to scan all data for alarm conditions at 15 Hz. This allows for potentially an enormous number of alarm messages to emanate from any local station, to an extent that any alarm display system can become overwhelmed. When the system being monitored is operational, and no alarm messages are forthcoming, it is comforting to know that the system is being watched very carefully, and nothing is being noticed that is out-of-limits. But when the system being monitored is not operational, an alarm screen can become so full that it is all but useless.

Various schemes have been suggested for overcoming this great disparity between what a human can interpret and what the control system can report. In the accelerator control system (Acnet), collections of devices are grouped together, such that entire sets of devices can be included or excluded from the alarm scan easily. In this way, a subsystem that is down for some period of time can be excluded from the alarm scan so that it does not contribute to filling up the alarm screen. Of course, the act of excluding a group of devices from the alarm scan brings with it a responsibility of later on including them when the subsystem is again considered operational.

This note discusses an idea that could be implemented in the local station system to support inclusion / exclusion of sets of devices from the alarm scan. The idea stemmed from an informal discussion with Harrison Prosper about reducing the alarm congestion for the D0 control system, in which the local stations play a major role. A special consideration for the D0 case is the slowness of access to the Hdb database, based upon Rdb from DEC.

General idea

Define a group of devices known to the local station via its local database. A control action is used to enable or disable the entire “family” of devices for alarm scanning. This control action itself affects a family device which can be in the alarm scan, serving to provide a alarm message reminder that the family is excluded from the alarm scan. The opposite control action restores the alarm scanning of the family of devices and also removes the reminder message.

Details of how it works

Consider analog channel devices only. Each analog channel's descriptor entry has a "family" word field. The value of this field is a "delta" channel number which, when added to the device's channel number, produces the channel number of the next member device of the family. The delta value can be positive or negative; thus, one can define an entire circular chain of devices that constitute a family. Beginning at any member of the chain, one can find all the members of the family. To do this easily from another system, there is a listype (#49) which can return the complete list of channel numbers of the family to which a given channel belongs.

This family implementation was originally designed to bring together all channels that relate to a V177 timer board; for example, a set of channels might be used to hold a clock event number that is selected to trigger the delay whose value is given by a different channel. It is limited in that there is only one family word per channel; thus, a channel can belong to at most a single family.

This proposal uses the family word to define a group of devices for the purpose of including or excluding them from the alarm scan. Due to the above limitation, such groups of devices must be distinct; a channel cannot be part of two different groups. Of course, family membership does not prevent any channel from being excluded from the alarm scan. One would want to define such groups such that they would normally all be included or all be excluded from the alarm scan.

The control action that would perform the group inclusion or exclusion could be supported by a new analog control type designed for this purpose. Each such family group of channels could include a special pseudo-channel that would be the target channel used to perform this function. Setting the channel to a nonzero value could cause the family members to be excluded from the alarm scan. The analog control processing would involve following the family chain and removing each such channel from the alarm scan, in such a way as to mark that this has been done, by setting another alarm flag bit, say. The special channel member, however, would not be removed. Also, its reading could be set to the number of devices in the family which were marked as removed from the scan. Its nominal and tolerance values could be set to 0, so that an alarm condition results from performing the exclusion control function when any member of the family is in the alarm scan.

Later, when restoration of the alarm scanning of the family is desired, a zero value could be used to set the special channel. The family chain would again be followed, and all channels which had been marked for temporary exclusion from the alarm scan would be restored to inclusion. The reading word of the channel would be set to zero, thus removing the alarm condition of the special channel. In this way, the special channel's alarm message serves as a reminder that some

family members are excluded that would normally be included.

As a refinement, the nonzero value used to set the special family channel could be a limit of the number of channels to scan in the family. But this may introduce an element of confusion that would be hard to recover from. Any nonzero value should probably cause a complete scan of the family.

Digital case

What about the digital alarms? The above discussion only addressed analog channels. Binary bits have no family word, so the same approach could not be used. But there is recent support for composite status words, which could be used to reduce the number of binary bit alarms. Collections of bits from up to 8 bytes can be grouped together into as many as 16 bits and assigned as the reading word of a channel. The nominal and tolerance words of that channel are marked to be treated by the alarm scan logic as a nominal digital pattern word and a mask word, according to a bit in the alarm flags word. In this way, one can get up to a 16:1 reduction in binary alarm messages.

In addition, these resultant channels can be included in the family assign ment above, so they can also be included / excluded along with other analog channels.

Conclusion

The scheme presented here is not the fancy AI-inspired approach to trimming down alarm messages that are displayed such that only the ones the viewer wants to know will be displayed. But it can be easily implemented and can reduce considerable alarm message congestion. It has a reminder feature so that excluded groups are not forgotten. It is also potentially very fast, as the group logic is managed locally. A host-level program would need to be written to manage the definition and display of the family groups.

FTPMAN Fast Time Data Addition

Up to 1 KHz data from IRM

Thu, Jun 23, 1994

The interface for Acnet Fast Time Plots is the FTPMAN support that is implemented by the local application FTPM in the local stations/IRMs. Support of data collection at higher rates than the current 15 Hz requires modifications to FTPM. This note discusses some of the problems and solutions relating to these modifications.

Current fast data support

In order to provide fast data support for the Macintosh, or other hosts using the Classic protocol, a new listype was developed that is described in the related "Moderately Fast Data Collection" document. Briefly, the listype uses an ident consisting of the channel# and event#. This allows access to data with times reported for the data points that are relative to a given event#, or to a given event# group. The internal ptr format includes the event# information as well as the offset within the 64kb array on the analog IP board of the last point reported. The requester's #bytes parameter and the return period parameter are used to determine the data point sample period. Data are supplied to the user's buffer mapped to this period. This scheme allows access to such data to mesh with the normal Classic request protocol paradigm.

Current FTPMAN support

The current support is limited to 15 Hz data collection. Until the use of the IP-based analog module in the IRM, no high speed data was provided by the hardware. Data at 15 Hz is taken from the data pool. The code runs at 15 Hz to capture this data and build up the answers for delivery at periods of 1–7 cycles at 15 Hz. Note that this support is still needed in IRMs for analog channels that do *not* come from the analog IP board. Derived signals, or other software-generated data, and all settings, are still limited to 15 Hz collection. The convention that is used to mark a channel in an IRM for this high speed ability is the channel number range chosen. Channels in the range 0100–013F are used for the analog board in slot *d*, and channels in the range 0140–017F are used for an expansion board in slot *c*. Channels outside these ranges are only accessible at 15 Hz. All channels on 133 boards are limited to 15 Hz access.

Time-stamps for FTPMAN

All data used by FTPMAN must be time-stamped. Event# 02, an event that occurs every 75 accelerator 15 Hz cycles, is used for this purpose. The units for the time values in FTPMAN data are 100 μ s, so that an unsigned 16-bit word is enough to specify this time value, using the range 0–50000 decimal. The current support for 15 Hz data uses values for this relative time accurate to only one 15 Hz cycle. For faster data, we must provide more accurate time-stamps.

The Classic fast data support develops times relative to any clock event decoded by the digital IP board, the latest version of which decodes all Tevatron clock events—except the 07 event used for 720 Hz timing. A 32-bit free-running 1 MHz counter is read every time an event interrupt occurs and stored in a table of times and elapsed-time-since-last-such-event indexed by event# 00–FF. This provides the basis for the precise time-stamps needed for up to 1 KHz fast data

collection. The hardware measures all 64 channels over 800 µ s every 1000 µ s, digitizing a channel every 12.5 µ s, allowing for analog multiplexer settling time and digitization.

It is desirable to borrow as much from the current fast data support as possible. The RFTData module is a “read-type” routine that supplies answers to a data request for the special fast-data listype# 82 decimal. It works from an array of internal ptrs that carry the information described above. Its answer result is a base longword of time in 10 µ s units since the chosen event, followed by the delta time between the last two such events in the same units, followed by (data, time) pairs of words, where the time values are relative to the initial longword in the same units. The user is expected to compute the time values needed for a plot by adding the base longword to the time word for each point. If the sum exceeds the delta time value, subtract the delta time value, as it means the current point occurred just after the latest event. This scheme avoids the need to supply a longword time value for each data point.

In order to use this scheme for FTPMAN, we must modify the data values. The internal ptr should always specify event 02, because that is the only one used by FTPMAN. (Actually, the client plot package can plot data relative to other events, but it demands only times relative to the 02 event from the front end.) The point values for FTPMAN need to be in (time, data) order, with the time word in 100 µ s units.

The internal ptr format is as follows:
(put picture here)
The Evt# will always be 02, and the FTPMan bit will be set, so that the RFTData routine will build the answer data in the proper format. The first two longwords need not be generated, and the point data is in the (time, data) order, where the time is in 100 µ s units. ChBlk selects which group of 64 channels.

Memory allocation

In the 15 Hz version of FTPMAN, the Pascal built-in procedure New was used, in which the memory allocated was fixed, sized for the maximum of 5 devices and 7 points per device. This won’t work for faster rates, because it would consume too much memory. But the Pascal New procedure does not work for variable size allocations. The new plan is to use Alloc, a function that takes an argument that is the number of bytes of memory to allocate. Its complement function is Free.

Server

From the server’s point-of-view, one can compute the data pointer structure based upon the number of data points that will be needed for each device. The protocol allows for each device to have a different sample period. The following formula will allow the server to compute the #points for each device and therefore the amount of memory required in the reply block for each device:

#points = (#cycles return period)*(1/15 second)/(sample period),

where the two times are in 10 μ s units. We use 1/15 second at Fermilab, where FTPMAN is needed. We need to be sure that the server and the non-server reach the same conclusions about the sizes of the data arrays, so it shouldn't depend upon the cycle rate of the server. This information can be computed before the memory is allocated for the reply message block. It can be placed into the reply block after the first continuous reply has been sent.

Because the sample period can vary for each device, RFTData can be called once for each fast data device. A fast-data device is one whose sample period is significantly less than 1/15 second. It can only be supported if the system is 162-based and has an analog IP board to support the fast digitizations.

When the server node receives replies from the non-server node(s), it uses the data pointers internal to the reply to find each device's data. For each one whose node# in the ident matches the node sending the data reply, copy the number of points indicated into the area reserved for it in the server's reply buffer. The number of points expected by the server should match the number of points indicated for that device in the non-server's reply message.

Timestamps

Timestamps are given for each data point in 100 μ s units, as noted earlier. For the 15 Hz data, this is done to 1/15 second precision using a scheme of synchronization through the network. Since that implementation, however, the FTPMAN client program has been changed so it supplies a starting value for a 1/15 second timestamp with the request message. When there is no analog IP board available, one may change the FTPMAN server logic so it takes advantage of this starting point, counting 15 Hz cycles to supply timestamps for the duration of the request.

For fast data rates, however, one should do better than 15 Hz precision timestamps. The digital IP boards that are a part of every 162-based system provide Tevatron clock event detection such that accurate timing of the 02 event is available. The timestamps are good to 10 μ s precision, so they are more than adequate for the task.

Non-server

For each device that is local, produce the required data. Again, the determination of the number of data points is done exactly as was done by the server. Devices that are not local are skipped. The number of points of data in the data pointers array may be zero. the server will ignore those devices whose node#s do not match the request.

With the array of internal ptrs, the answers for each fast data device can be updated with a call to the RFTData routine. But the data that results will be modified for the FTPMAN format requirement of (time, data) order and time words in 100 μ s units relative to the 02 clock event. The two initial longwords produced by RFTData can be used for the modification logic, but they will not be included in the data reurned by FTPMAN. It is logical that FTPMAN should perform this modification, rather than RFTData itself.

Internal Ptrs

Data request object code

Oct 6, 1989

When a data request is initialized, the array of idents for each listype in the request is "compiled" into an array of Internal Ptrs in order to more efficiently update the answers. This translation is done even for a one-shot request.

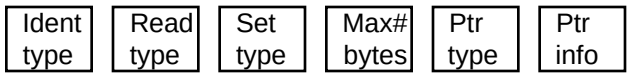
There are two variations of data requests. The first is a data request that originates from a Local Station (either for the current application page or in response to a Data Server request over the network). Such a request may include data from various nodes; the local station selects out the non-local idents from the request to form an "external request" that is sent to the network. Each node receiving the network request responds to its part of the request only, and the various "answer fragments" are put together in original request order to build the response to the original request. The module that processes such a request is REQD, and the module that includes the "compiling loops" is REQDGENP, because it generates the internal ptrs for an REQD-style request. The REQD module is the "odds-on favorite" for having the most complicated logic in the system code. REQDGENP is a subroutine called by REQD and is therefore an extension of REQD. As such, it inherits much of REQD's complexity.

The other variation of data request is the "ordinary" network data request. In this case, the request may include references to various nodes, but the processing of the request will result in building only the local node's answer fragment. (Obviously, it is this ordinary network request that is used when a node processes the first data request variation.) The ordinary network request is processed by the PREQD module which calls PREQDGEN as a subroutine. The code for this version of the "compiling loops" is simpler than that in REQDGENP, because references to data from other nodes can be ignored, and an external request does not have to be built.

Since there are two variations of data requests processed by two different routines (REQDATA and PREQDATA), which in turn invoke two different subroutines (REQDGENP and PREQDGEN), it is necessary to write two different "compiling loops" when a new type of internal ptr must be generated.

Listype table

The format of a listype entry is as follows:



The *ident type* byte is a small index into the ident table the gives the (short) length of the ident. It is also used to insure that a request which specifies more than one listype uses only listypes that are ident-compatible, which just means that all listypes in the request use the same ident type.

The *read type* byte is an index into the READS branch table in the COLLECT module and thus selects the read type routine that produces the answers to a data request given the array of internal ptrs corresponding to the array of idents in the original request.

The *set type* byte is an index into the SETS branch table in the SETDATA module and thus selects the set type routine that accomplishes a setting action given the ident and the accompanying data. The *max#* byte is the maximum number of bytes of setting data that are acceptable for that listype.

The *ptr type* byte is of two varieties. If its value is < 32, then it is a table#, and the *ptr info* byte is an offset to a field of an entry; if it is ≥ 32, the ptr type byte is used to index into the GENS table in the appropriate module (REQDGENP or PREQDGEN) to select the ptr type routine that produces an array of internal ptrs given an array of idents.

REQDGENP compiling loop

The register-based call to the ptr type routine is as follows:

D2.L= Ext ident cntr in hi word, #idents (bit#15 = long id flag) in lo word
D4.W= Ptr info byte
D5.B= local node#
D6.W= #bytes of data requested from each ident
A1.L= Ptr to array of idents
A2.L= Ptr to output array of internal ptrs
A3.L= Ptr into array of external idents

Upon exit, A2 should be advanced reflecting the number of internal ptrs that have been stored. The number of external idents is incremented in the upper word of D2 as they are encountered. For each such ident, the ident itself is copied into the array of external idents pointed to by A3. This is used in the external data request that will be sent to the network to collect the answer fragments from the other node(s) referenced in the request. Other registers are scratch except, of course, A5/A6.

PREQDGEN compiling loop

This case is simpler from the previous case since all idents which reference data from other nodes can be ignored. The register-based call to the ptr type routine is as follows:

D2.W= #idents (bit#15 = long ident flag)
D4.W= Ptr info byte
D5.B= local node#
D6.W= #bytes of data requested from each ident
A1.L= Ptr to array of idents
A2.L= Ptr to output array of internal ptrs

Upon exit, A2 should be advanced reflecting the number of internal ptrs that have been stored. Other registers are scratch except A3/A5/A6.

Internal ptr types

Ptr type < 32. Table entry field ptr

If entry# out-of-range, use ptr to zeros, else use ptr to field in table entry.

Ptr type = 32. Memory address ident

For short ident case, map 24 bits into 32 bits by using `$FF` for the hi byte if the 24-bit value is ≥ `$F00000`, else use `$00` for the hi byte. For the long ident case, use the full 32-bit address for the internal ptr.

Ptr type = 33. Device name to channel ident

There is no long ident case, as the ident is merely the 6-character name. The internal ptr is the node# in the hi word and the channel# in the lo word.

Ptr type = 34. Binary status via Bit ident

Use the lo 24 bits of the internal ptr to store the address of the `BBYTE` entry which contains the value of the status byte. The hi 8 bits contains the bit# in the range 0–7 of the bit in the byte to be sampled.

Ptr type = 35. Global variables (relative to A5)

The internal ptr is merely the address of the global variable.

Ptr type = 36. Generally interesting data

The internal ptr is merely the address of the byte in the `G.I.D.` pool.

Ptr type = 37. Serial input queue data

The internal ptr is the base address of the serial input queue. If there were more than one serial port, it would be logical to use the base address of the serial input queue associated with the port according to the ident value.

Ptr type = 38. Data stream data

The data stream index is in the hi word and the ptr info is in the lo word. The ptr info (and hence the listype#) can be used to identify whether old data is being included in the request.

External answer buffer ptrs

For the local or data server types of requests, the internal ptr may be marked as one which points into an external answer buffer. Normally, bit #31 is used for this purpose. The read type routine, when it sees the sign bit set, will simply copy the bytes of memory pointed to by the other 31 bits; any special processing to produce the data was already done by the node that sent the answer fragment that is copied by the Network Task into the external answer buffer. In the case of a memory address used by read types #1 or #4, where the job of the read routine is merely to copy the memory pointed to by the address, this special use of bit #31 is not used, as the job is to copy memory bytes in any case, no matter what it points to.

For writing new ptr type routines, it is advised to study some of the seven current examples for ptr types 32–38 above. Recall that two routines should be written, one referenced from `REQDGENP` and the other from `PREQDGEN`. The latter type is simpler than the former.

MMAPS table entries for D0 Boards

Nov 12, 1991

Central Tracker FADC Memory Map (map-type #1)

D0D0	0006	0010	0000	Loop over 5 commands 16 times
0001	0022	000E	0000	t1, t2, t3, t4, s1, s2, s3
0001	0004	0002	0000	depth
0001	0006	0002	0000	chanNumber
0001	0002	0004	0000	dacGain1, dacGain2
0001	0002	0002	0000	dacOffset
0001	0038	0000	0000	Skip to end of 128-byte block

Total data: (14+2+2+4+2)*16= 384 bytes

Calorimeter ADC Memory Map (map-type #2)

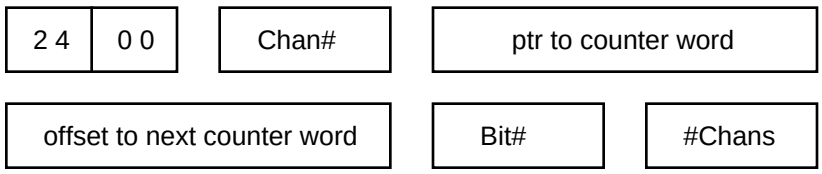
D0D0	0002	0020	0000	Loop over next 2 commands 32 times
0001	0000	0060	0000	Access first 96 bytes
0001	0020	0000	0000	Skip 32 bytes to end of block

Total data: 96*32= 3072 bytes

Data Access Table function
Thu, Feb 18, 1993

This note describes a Data Access Table routine that monitors 16-bit counters to insure that co-processors are functioning normally. It can also monitor the rate of counter advance to show load level. By including a result status bit in the alarm scan, one can generate an alarm message when either a co-processor quits working, or access to a co-processor stops working.

Data Access Table entry layout:



The change in value of the counter word since the last time this entry was processed is stored as the reading of the given Chan# channel. This can be one 15 Hz cycle, or it can be more using an appropriate $7F period entry to specify a sub-multiple rate of execution. For #Chans > 1, the offset longword is used to advance the ptr to get the next counter word address, which is then used to target the next channel.

An optional Bit# word specifies a Bit# that is set or cleared to indicate whether the change difference value is nonzero or not, respectively. (If this option is not used, the Bit# word should be zero.) The reading of this Bit can then be used to generate an alarm message when it is zero, indicating that the counter is not changing. This feature is probably easier than trying to predict what the value of the change should be in order to set the nominal and tolerance values to alarm on the analog channel.

If a bus error occurs when accessing the counter word, the delta value is set to zero, and the (optional) Bit is cleared, indicating that the counter is not changing.

Note that the longword offset value allows accessing multiple co-processor counter words across the vertical interconnect with a single entry, if the counter words are in the same location in each co-processor's memory.

MPW Conversion

Development platform for local station system software

Jan 13, 1992

During a ten day period around New Year's Day 1992, the local station system software was ported to the MPW development system used on the Macintosh. This brief note describes something of how the process went. Reasons for the change from the Octal tools on the Vax used for many years are these:

1. Octal software is no longer actively supported. The 68020 assembler seemed to be unable to generate the 32-bit PC-relative addressing mode that is needed to get an address of a routine or table in the system code in a position-independent way in a large (> 32K) program.

2. The Pascal compiler pre-dates the floating point co-processor chip and cannot generate the instructions for type Real support. Local applications that need such instructions were developed with MPW Pascal.

3. The size of the system code (60K) was stretching to the limit a model of programming conventions that would theoretically break at 64K, as it attempted to maintain the limits of 32K branches of the 68000 CPU. Since all local stations now run on 68020 CPU's, this restriction can be eliminated.

4. The Macintosh is a conveniently accessible platform if one has a Macintosh on his desk. It is actively supported by Apple Computer for its own software development and for Macintosh developers. The cost is $525 for a single user, with support for both C and Pascal.

5. The most recent version of MPW, version 3.2, allows support of "32-bit everything" including the ability to generate a code resource >32K bytes. The local station system code is linked as a single code block segment.

6. The MPW assembler supports structured assembly syntax through an extensive set of macros. Many other assemblers available today do not include support for structured syntax (IF-THEN-ELSE, REPEAT-UNTIL, etc). The local station system software uses this syntax heavily. Elimination of that syntax from the source code is inconceivable.

The first job to be handled was to find a way to automatically change the structured assembly syntax from the format Octal supported (and Motorola in their original EXORmacs development system) into the format used by MPW. The MPW shell supports Unix-like (but not fully Unix-compatible) commands for editing that include "regular expressions" for indicating the text to match for a Replace command. A set of 20 Replace commands were designed for use via a command file to do the structured syntax conversion. Without this automatic aid, the job

Each of the approximately 100 source files of 31K lines of assembly code was passed through the automatic conversion first and boiler-plate header lines inserted to invoke the structured syntax macro support and the system constants "include" file. Then each file was checked for details that had to be changed manually. This included identifying code sections individually with the PROC directive and data structure templates using the RECORD directive. There were also some pathological cases converted incorrectly by the automatic process.

Then an assembly was attempted. Sometimes no errors resulted. Sometimes up to 10 pages of errors were listed on the screen. In the latter case, usually a few error corrections eliminated many others. Eventually, all errors were gone, and it was time to move on to the next one. After the first few dozen files were converted, things began to go faster.

After all source files were converted and assembled, the next step was to link them into a system. At this point MPW 3.2 was needed to get the "-model far" option that permits building a large CODE resource. After correcting the errors in undefined names, a 60K CODE resource was in hand.

The next step was to use the Hex tool to translate into S-records to be used for downloading via the serial port into a VME local station. This tool was written by someone from England long before, and it was limited to CODE resources of 32K bytes. Inspection of the Pascal source code revealed that the buffer was only 32K in size. The size was changed to 100000 (decimal), and the tool was re-compiled with the "-model far" option used for both the compiler and linker commands in the "make" file. It worked.

The S-records were downloaded into local station 0576, and after a reset, the system came up running. But a bug was found that produced a "3" error (internal inconsistency) on the memory dump page. Upon correction, the system ran apparently ok. In the next days, a few more errors were found and corrected, including the macro file FlowCtlMacs that supports the structured syntax.

Name Table

What's in a name?

Wed, Sep 9, 1992

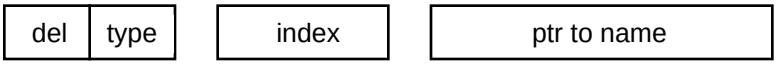
The VME local station systems have names that are used for analog channels. They may also have names for binary status bits. For a station with 2K analog channels, the time to execute the current linear search for a 6-character name can be 5 ms. This note describes an imple mentation of a generalized name table used for fast searching of names using a double hashing algorithm.

Initialization

At reset time a table is built that contains all the known names in the system. Each system table (in non-volatile memory) that houses names of anything is scanned, and the names are entered into a hashing table. Let each name length be even and up to 32 characters in length. The name is reduced to a longword and divided by the number of allocated table entries which is the *larger* prime number of a prime pair. In this way, the quotient is an index into the hashing table. If that entry is examined and found *not* empty, it is tested for a match against the search name. If there is a match, the name is already in the table. If there is no match, the next entry in sequence is examined, using as a sequencing delta a value obtained by dividing the same dividend by the *smaller* of the prime pair and adding 1. (This scheme is called double hashing, as it uses two hashing functions; it is described by Knuth in his volume on Sorting and Searching.) This matching procedure is continued in this way until an empty entry is found. Enter the new data into the empty entry. Only when the table is full will there be no empty entry found, although the efficiency of this scheme falls off somewhat before that happens.

Hash table entries

Suppose the information in the entry is 8 bytes as follows:

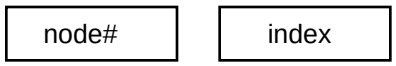


The del byte is a "deleted" flag. The type byte denotes the name type category, allowing for names of different types to match. The index word is the result data of a name lookup of a given type. (The most obvious example is an analog chan nel number.) The ptr to name is the address of a name field in a system table entry. To allocate room for 8K names, 64K bytes is required for the table. It should be made rather larger than this to reduce the likelihood of collisions.

To process a request that specifies a name ident, the name table is consulted. If the hash code points to a non-empty entry, match against the name field pointed to. If there is no match, continue with the next entry until a match or an an empty entry is found. In the latter case, there is no match on the name, so the name does

not exist. If a match is found, then the index word is returned to the requester.

The returned data has this format:



Insertions and deletions

To enter a new name into the table, follow the procedure described under Initialization. To delete an entry, the same sequence is followed, but upon finding an match, mark the entry deleted. It cannot me marked empty, since doing so may mask further entries in the chain.

```
Function NTLookup(VAR name: NameType; lng, nameType: Integer;
        VAR index: Integer): Integer;

Function NTInsert(VAR name: NameType; lng, nameType: Integer;
        index: Integer): Integer;

Function NTDelete(VAR name: NameType; lng, nameType: Integer): Integer;
```

Error return codes:	Routine:	
0	No errors	
–1	Invalid name table (All)	All
–2	#chars must be even and in range 2–32 (All)	All
–3	Name not in table	`NTLookup, NTDelete`
–4	Table overflow	`NTInsert`
–5	Duplicate name already in table	`NTInsert`
–6	Bus error. Bad ptr in table entry	All

Name changes

When a setting to an analog descriptor is made, a special check is made to determine whether the setting will result in changing the name field for that channel. If so, the current name is deleted and the new name added. This can cause table entries to be used up as entries are deleted, but they can be re-used. At reset time, rebuilding the table removes such “deleted” entries.

Name table header layout

NTKEY	NTSTRT	NTMSZ	NTVAC
NTREQS	NTCOLL	NTMAXC	NTLAST
NTFOUND	NTDELET	—	NTTIMZ
NTCNTR	NTDUPC	NTDUPL	NTTIME
NTCNT16	NTDCH16	NTDUP16	NTTIM16
—	—	—	—
—	—	—	—
—	—	—	NTTIMT

```
NTKEY    Name Table Key 'NT'
NTSTRT   Offset to first entry
NTMSZ    Maximum #entries in table
NTVAC    # Vacant entries in table
NTREQS   # times NameSrch called (diag)
NTCOLL   # collisions during NameSrch (diag)
NTMAXC   Max # collisions for a name (diag)
NTLAST   Last entry inserted (diagnostic)
NTFOUND  Offset to last name match
NTDELET  #entries deleted
—        spare(1)
NTTIMZ   Time to clear table during NTINTZ
NTCNTR   Count of 6-char names inserted
NTDUPC   Last duplicate 6-char name found
NTDUPL   Count of duplicate names
NTTIME   Time to enter all 6-char names (0.5 ms)
NTCNT16  Count of 16-char names inserted
NTDCH16  Last duplicate 16-char name found
NTDUP16  Count of duplicate 16-char names
NTTIM16  Time to enter 16-char names (0.5 ms)
—        spare(11)
NTTIMT   Total time to initialize name table
```

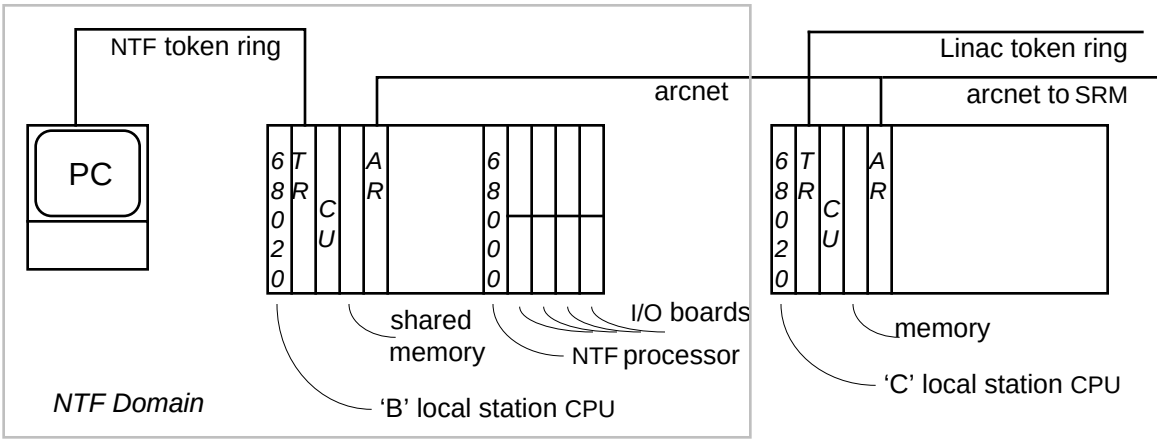
NTF Connection

Linac control system implementation

Dec 20, 1991

NTF architecture

Neutron therapy signals are primarily interfaced via a special 68000 CPU board running a small (4K) program that measures the delivered dose and stops the treatment when the prescribed limit is reached. It also checks for a number of error conditions and stops treatment if these occur. This dedicated program is driven by a PC-based user interface that is connected to a local station whose crate houses the dedicated processor.



The 'B' local station supports the token ring connection to the PC that is the NTF host, providing the user interface for patient treatment. This local station has the same complement of hardware/software as any other, with the addition of a special co-processor to handle the NTF-specific functions. The non-volatile memory board that houses the local station database and software also provides the connection between the two processors in the same VME crate, which is necessary because the I/O boards are accessible only from the MVME-110 68000 board. Every 15 Hz cycle, the 68000 copies about 256 bytes of data readings and diagnostic info into (a small part of) the shared memory. Likewise, settings are sent to the 68000 via a small command message queue in the shared memory. Station 'B's know ledge of the shared memory is limited to the data access table entries that copy from the shared memory data and also the co-proc essor queue table entry that specifies the location and size of the command queue.

The ‘C’ local station provides access to the NTF system from the rest of the control system in a controlled way. It collects data via arcnet from ‘B’ using the SRM data acquisition protocol. It also makes settings as if ‘B’ were an SRM.

Note that there is some effort to isolate the special NTF functions from arbitrary access by the rest of the control system; however, the rest of the control system does need to access some NTF data and even control a few signals. The ‘C’ local station and its arcnet connection to ‘B’ provides for this.

History

The previous version of the architecture used a byte-wide fifo link connection between the special NTF co-processor and station ‘C’. The reason for the new arch itecture using arcnet is to retire the special byte-wide link interface, replacing it with one commonly used with new Linac local stations.

Data acquisition

In order to make station ‘B’ behave to station ‘C’ as if it were an SRM, a local application called SRMD is installed in station ‘B’. It receives the SRM cycle request issued by ‘C’ every 15 Hz cycle. (In fact, it is broadcast to the arcnet network so that both its SRM and ‘B’ receive the same request message.)

By the time ‘C’ receives its (delayed) 15 Hz interrupt signal, the data from the special NTF processor has already been copied into the shared memory. This means that the request message subsequently sent by ‘C’ cannot arrive too early for ‘B’. When ‘B’ is processing its data access table, it calls the SRMD local appli cation, which looks for the request message over arcnet from ‘C’. (SRMD may wait for a short time to be sure ‘C’ has a chance to issue that request message.) The arcnet interrupt code in ‘B’ should expect a cycle data request and pass it through to the message queue that SRMD polls. (This is also done for data acquisition replies, as in ‘C’.)

The local application builds a reply message containing readings of selected ana log channels of station ‘B’, including 8 words of nominal and tol erance values from the co-processor’s prom-based table used for check ing beam transmission ratios. (These latter values do not change, of course, but they are needed for operator reference.) The reply message is queued to the net work and sent to ‘C’, where it is mapped into assigned channel readings in its database. Other nodes on token ring can access this data from ‘C’ to view NTF data.

Settings

For handling settings, a node sends a setting message to ‘C’. For a channel in the NTF system which is permitted to have control, a short setting message is sent via arcnet to ‘B’. The SRMD local application reads this setting message containing a reference to an analog channel number of ‘B’. It merely calls the setting routine to control that “local” channel. If the analog control field of the channel’s analog descriptor entry permits settings, a short setting message is placed into the com mand queue. The co-processor notices it and executes the command. Assurance of success comes from the expected change in the NTF data refreshed every cycle.

Other details

The PC host operates through ‘B’, whereas other nodes’ access to NTF is via ‘C’. This provides an easy way to limit which signals are controllable from outside the NTF domain.

Currently, NTF signals controllable via ‘B’ are the beam on-off pulse ratio, the dose accumulations and dose limits, un-clamp integrator gate, and reset. Signals controllable via ‘C’ are the beam on-off pulse ratio, nominal and tolerance values for the 58° and 32° magnet currents, rise time of the 58° magnet, and reset.

Because ‘B’ behaves as an SRM to ‘C’, there is a status bit that can be monitored to generate an alarm condition if ‘B’ is not responding to the request message. Data acquisition from each SRM includes generation of this pseudo status bit.

NTF Interlocks Checking

What do the little LEDs mean?

Feb 15, 1992

The NTF control system includes a special co-processor that shares a VME crate with the NTF local station. This 68000 cpu board runs a small program which performs certain NTF-specific checks relating to neutron beam delivery. This note describes these interlock checks the results of which can turn off the NTF beam and indicate the reason via a set of eight LEDs. These lights can be seen at the NTF station, but they are also part of the datapool that is available from node 061C via the Arcnet connection. (See the document "NTF Connection" for a more extensive discussion.)

- Missing beam check

Two toroids are used to measure beam for NTF. CTOR1 measures the beam that exits Linac tank 4, and CTOR2 measures the beam that is headed for the NTF target that produces the neutrons. There is a constant beam threshold that is used to discriminate whether either toroid reading represents *any* beam. That threshold, which is a program constant, is currently a value of 0.31 volts, corresponding to 3.1 ma using the current 100 ma fullscale value for those signals. Any reading less than this value is considered noise that is interpreted as no beam.

Both toroids are examined (in absolute value) every cycle. Only if *both* are below the threshold value, the pulse is ignored for purposes of determining missing beam status. If CTOR1 is above threshold, there is beam and CTOR2 is compared against a value of 0.68*CTOR1. If it is less, the transmission "around the bend" is considered too low, and the pulse is considered a "bad" beam pulse; if it is greater, then the pulse is a "good" beam pulse. After 15 valid beam pulses, if there are less than 8 "good" pulses out of 15, then Missing beam status results.

In the unlikely case that the reading of CTOR1 is below threshold, and the reading at CTOR2 is above the threshold, the pulse is counted as a "bad" beam pulse. In this case, the "missing beam" refers to inconsistent readings, as CTOR2 would not be expected to exceed CTOR1.

The missing beam logic does not depend on the ON-OFF sequencing described below. The determination is made after every 15 valid beam pulses.

• X1,X2,QP limit checks

Long term accumulations are checked following the end of each ON sequence. Normally, the practice has been to program a sequence of 5 ON pulses followed by 0 OFF pulses, which just means that the following checks are made every 5 beam pulses. (Another detail relating to the #pulses in a sequence is that the long term accumulations for beam charge and ion chamber integrations are accum ulated in a 16-bit word during the sequence. Since they are 12-bit resolution readings, keeping the #ON pulses below insures that there can be no overflow.) Note that a beam pulse here means any *scheduled* beam pulse, not one that is required to exceed the threshold described in the Missing beam section.

Each long term accumulation is compared against the established limit value. (See the document "What are the Units of X1 LIMIT" for more details.) If the accumulation exceeds the limit, then the corresponding status results. This is the normal way that neutron dose treatments are terminated. (There are also hardware limits that are set slightly higher, including a time limit, as an extra measure of safety.)

• QP/X1, X2/X1 ratio check

Ratio checks are made at the end of each ON sequence. The reference values for these ratios are stored in PROM and thus cannot be easily changed. This is intentional, for they are used to insure that the beam transport and neutron production are functioning normally as established via careful calibration. In the PROM are nominal and tolerance values for each ratio—and for each reference voltage as described below. If a ratio falls outside the tolerance window, the corresponding ratio status results.

• V1,V2 Reference voltage checks

Reference voltages are checked at the end of each ON sequence. They are a measurement of the power supply voltages used for the integrators of the ion chamber signals. The nominal and tolerance values are kept in PROM. If a ref erence voltage falls outside the tolerance window, a voltage check status results.

• LED lights

All eight checks are reported via a set of LEDs and also as data that can be mon itored. Any LED that is set will inhibit further beam until CTF RESET is asserted.

Ptr-type and Read-type routines

Wed, Feb 22, 1995

Data request support for local stations/IRMs uses “ptr-type” routines to “compile” a request for data specified via a listype#, where the request in this context is comprised of an array of “idents.” A ptr-type routine scans the array of idents and produces an array of “internal ptrs” that represents a kind of “object code” for the original request. This is done so that “read-type” routines can generate the reply data most efficiently, especially for the case of requests for periodic replies.

To illustrate this simply, consider a request for readings of analog channels. The listype# for analog readings is 0. The idents used for such a request are channel numbers, each given as a two-word structure, the first word being a node# and the second word a channel#. The analog data pool of a local station is organized as a simple array of records, one field of such records being used for the reading word value. The format of internal ptr used in this case is a pointer to the given channel;'s reading field in its analog data pool record. The update logic, then, which utilizes a read-type routine, is a simple loop that gets an internal ptr from the array and dereferences it to obtain the reading value result, and loops over the number of idents—the same as the number of internal ptrs—presented in the original request.

For the Classic protocol, as used by local or page applications, a data request is made specifying an array of listypes and an array of idents. If more than one listype is used, each must be associated with the same ident types, as the same ident array is “compiled” into internal ptrs for each listype given in the request. Sometime after the request has been made, the application invokes Collect to retrieve the results. Typically, an application does this on the next operating cycle. If the data should not yet be available, because the request required data to be retrieved from another node, then the Collect routine waits for the external data to arrive. There is a timeout for this that is normally 50 ms after the current cycle.

There are only a few error return codes that Collect returns, as follows:

0	No errors.
1	Invalid list#. Maybe the request was really strange.
2	Data not yet available. Collect called on same cycle request was made.
3	Internal corruption of request context.
4	Bus error detected in collecting answers.
5	Too few bytes received from an external node
6	Too many bytes received from an external node
7	Answers tardy, but have received answers at least once from ext node.
8	Answers tardy. Nothing ever received from ext node.

Note that the only nonzero error codes normally seen are 4, 7, 8. Ask for memory data using an address ident that is invalid for that station's hardware, and an error 4 is returned. Ask for data from an invalid node# or a valid, but non-operating node#, and an 8 will be returned. Ask for data from a node that is operating at a rate slower than the requesting node, and intermittant 7 errors will appear. During a periodic request activity, if that node drops off the network, solid 7 errors will occur.

The Ptr-type routines that build internal ptrs try to do so without error. If a system table does not exist, for example, the request cannot reasonably be supported by this node, so zeros are returned for the answer to the request. The ptr-type routine and the corresponding read-type routine for the given listype work together to produce the answer results. The internal ptr is the output of the ptr-type routine and the input of the read-type routine. There is no mechanism for returning an error code in the call to the data request procedure. The only error return that a read-type routine can return to Collect is a bus error indicator, which causes Collect to return error code 4. One might say that the attitude here is that a faulty request results in answer data of zeros, by definition.

In writing a ptr-type routine, consideration must be given to what is needed by the corresponding read-type routine to define the answer data. In the case of an ident that is from an external node, an external answer ptr form is used that is usually a ptr to the proper place in that external node's dedicated reply buffer for that request, which is allocated as part of the request block data structure for that request. In order to indicate that case of an internal ptr value, the sign bit is set in the internal ptr, which is a 32-bit longword. This means that typically one will find in a read-type routine corresponding logic that detects the presence of the sign bit to signal an external ptr case, removes the sign bit, then uses the low 31 bits as a ptr into the proper place in the external node's reply buffer in the request block. (All dynamic memory, and hence all request blocks, reside at memory addresses below 1 MB, so nothing is lost by commandeering bit# 31 for this purpose.

The case of a memory address ident is special. In this case, the user specifies an arbitrary 32-bit memory address from which data is to be retrieved. So we must use all 32 bits of the internal ptr for the given memory address. So in this case, the ptr-type routine does NOT set bit# 31 of the internal ptr to mark the external answer ptr case. And the read-type routine merely copies memory data from the address given. If that address happens to point to the external answer buffer, memory is copied from that buffer. If it happens to point to a field in a system table entry, memory is copied from there. In either case, memory is copied to produce the answers. It does not matter from whence it is copied.

RDATA Periodicity

How to access data infrequently

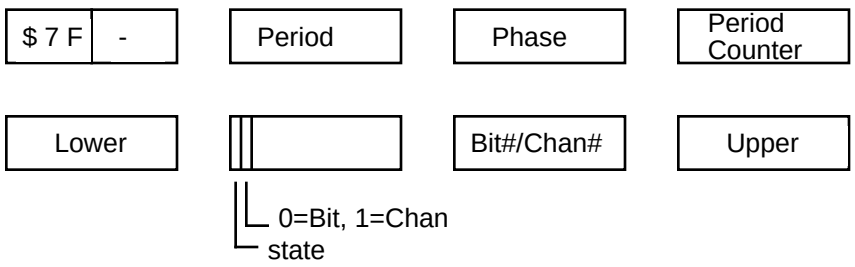
Sep 14, 1989

The Read Data Access Table (RDATA) has traditionally been scanned on each 15 Hz cycle to read all the data from the hardware interfaces into the system's datapool. The analog data is stored in the ADATA table and the binary status data is stored in the BBYTE table. But there can be reasons why the data should not all be read at 15 Hz. One reason may be that it takes too much time to read everything every cycle. Another reason is that the very act of reading some hardware interfaces can cause noise interference. The liquid argon temperatures in D0 are an example of the latter case. This note describes a plan for accommodating access to data at more leisurely intervals.

In order to arrange for less frequent execution of some entries in the RDATA table, some means of keeping a counter is required. We can assume that the table will be scanned at 15 Hz, but some entries may not be executed each time. To pick a concrete example, let's say that we want to read some hardware at 15 Hz and some other hardware at 1 Hz, but there is a *lot* of hardware that must be read at 1 Hz. We want to schedule the accessing of some of that hardware on different cycles. This means that we want to specify a "phase" value for sequencing the components of the large amount of 1 Hz data. So we need a way to specify a period, a counter to count out the period value, and a "phase" value to initialize this period counter at reset time.

Consider groups of entries in RDATA which should be processed with the same period and phase. Use a special entry which acts as a sort of header for the following entries and specifies this period information for the following group of entries. The group size includes all entries until the next special period-holding entry or the end of the table, whichever comes first.

Suppose we use the following period-holding format:



Note that the period is specified as a 16-bit field. This allows for periodic data collection to be as infrequent as approximately every hour. The optional Bit# or Chan# and state value (in the sign bit of the word) can be used to condition the processing of the entire group of RDATA entries which follow this period entry. The Bit# / Chan# must be nonzero to enable this option. When using

this feature, one must take care to have collected the binary status data or the analog channel reading data, on which the Bit# / Chan# test depends, *before* the period entry which needs to reference it.

For the Bit# case, the following group of RDATA entries is enabled when the state of the indicated Bit matches the state bit. For the Chan# case, when the state bit=0, the group is enabled when the reading of the indicated Chan is within the range of values given by the Lower and Upper words. If the state bit=1, the group is enabled when the reading is outside the range.

At reset time, the system scans the entries in the RDATA table for these period-holding entries. For each one it copies the phase value into the period counter. (If the phase value is larger than the period value, it merely zeros the period counter.)

At 15 Hz time, when the Update task is reading the data from the hardware, the RDATA table is scanned for a period-holding entry. All entries are ignored until the first such entry. If the Bit# / Chan# word is nonzero, apply the indicated test above. If the test fails, set a flag to ignore the following entries until the next period entry; if it is successful, clear the period counter. Next, if the period counter is zero, copy the period value into the period counter and set a flag to enable processing of the following entries. In any case, decrement the period counter. For the simple case of 15 Hz, the period would be 1, and the phase would be 0. Viewed on the memory dump page, the period counter would seem to always have the value 0. For longer periods, the phase may be set to any value from 0 up to the period - 1, and the period counter would exhibit its decrementing behavior.

This capability for execution of various accesses to hardware interfaces on an infrequent basis can also be used for other infrequent scheduling within the VME system. The meaning of each entry type in the RDATA table is fairly open-ended, and new code can be added to do other jobs.

Readings Averaging

Accelerator protocol feature for local stations

Sep 15, 1991

Introduction

The Linac operates at 15 Hz, and each cycle may or may not produce beam, according to the accelerator timing system clock events. For normal viewing of Linac device readings, as on a parameter page, it has been traditional for many years to display reading values averaged over beam pulses only, ignoring the pulses which have no beam. In the case that there are *no* beam pulses occurring in the averaging interval, then the average of readings for *all* cycles in the interval is shown. The averaging logic serves two purposes. The first is that data from beam pulses is selected preferentially over that from non-beam pulses. The second is that fluctuations are reduced in the displayed values.

The averaging logic has always been done by the application program that needs the values. In order to do it, the application must be aware of which cycles are beam cycles. This information is available at each Linac local station by a status bit that is wired to all stations. Note that this signal indicates that beam is *scheduled* to be delivered by the Linac; it does not *guarantee* that protons will in fact be accelerated. But it serves to provide a proper value to display beam-related data, such as beam current toroid readings. If a beam current reading were used in place of a status bit value to determine whether a pulse was a beam pulse, then it would not display correctly under conditions of low beam currents below the threshold value used for that determination.

The local station parameter page (on the small consoles) checks its own local beam status bit to decide whether the present cycle is a beam cycle. The Macintosh parameter page (written by Bob Peters) uses a pseudo-channel value from a specific local station that contains the same information. The Vax parameter page (written by Jim Smedinghoff) uses a beam toroid reading.

When the beam cycle status information is delivered over the network, rather than measured locally as in the local stations themselves, there is an additional problem of correlation with the present cycle's data. If the data to be averaged comes from a different source node than the beam status, one must insure that the beam status is known from the same cycle; otherwise, the average value may be diluted, especially for beam toroid readings. The Macintosh parameter page uses the data server for its requests, so the pseudo-channel reading is delivered in the same message along with the data to be averaged.

In case of the Vax parameter page, the data may not be deliverable in a single message until the entire Linac controls upgrade is in place, and all Linac data is requested from the Linac data server node. In the interim, while some Linac data

comes via the PDP-11 front end and some comes from the server node or from a specific node, correlating the data from different sources may be difficult. This is especially true because the Vax data pool manager (DPM) does not provide support for 15 Hz correlated data, even when it *does* come from a single node; a requester only gets (via DPGET) the last value of a given data item that was received. Another data item is obtained by a separate call to DPGET, and there is no guarantee that another cycle's data message has not been received since the first call. In fairness, it should be pointed out that one can obtain a sequence number for each item to determine whether new data has been received, but there is no way to insure that one can capture correlated data, even when the network actually delivers it. For Linac studies, this is a serious limitation of the accelerator control system, in the opinion of this writer.

In part because of the difficulties mentioned above, people have several times requested that the local stations perform the averaging logic. I have resisted this suggestion in the past because it seems to me that it amounts to placing application-specific code in the local station. One does not want to reach a point where applications can only be written on the Vax by adding application-dependent code to the local station system software as well. The local station's main job is to deliver the data to any and all requesters; what a requester *does* with that data should be up to the application program completely. In this case, however, the reality of the situation is that the Vax software is not designed to retrieve correlated 15 Hz data. If support for average data readings is to be provided to Vax applications, such as the parameter page, it may *have* to be done by the local station software.

Implementation

Although the averaging logic is well-known, it does not easily fit into local station support for data requests. The idea of averaging all data readings in the local station is rejected *a priori*; therefore, the support should be provided on a request basis. How does one specify to the local station that average values are needed? One suggestion is that a request period which is neither one-shot nor 15 Hz would be the indicator that average values are desired. The notion here is that sampling data from a 15 Hz Linac at other than 15 Hz is at best a hit-or-miss proposition and not for serious data-taking. Of course, given the averaging logic as stated above, the result of "averaging" data readings from a single pulse must be the same as the single reading from that pulse, independent of whether it was a beam pulse. This means that all readings could be treated with the same logic without regard for the data request period.

For the accelerator data request protocol supported by the RETDAT network task logic, which is the protocol used by the Vax consoles, each device request packet included in a data request message contains an SSDN. Inside the SSDN is a listype#, the fundamental data type specification used for local station data

requests. In order to provide reading values useful for plotting as well as as for display without making up new names for accessing device averaged readings, the same listype# should be used. To detect the need for averaging data in a request, a scan must be made for the use of the readings listype# (which = 0) in any data request packet. Also, the check could be made for the reading property index where two bytes of data are requested. Note that we only want to average analog readings. There is a potential problem with using the readings listype# as a key. We will be providing reading words as basic status values that contain collections of status bits. This will be done to provide a name and alarm mask for such assembled status words. We should therefore include the check on the reading property.

Internally, the averaging requires additional storage for the accumulation of reading values over the beam cycles (or the non-beam cycles in the absence of beam cycles) for each device for which a reading is requested.

The request period is used to specify the averaging interval. This interval is not synchronized with anything but 15 Hz cycles. For the Linac which often delivers 13 successive pulses of beam to the Booster, the averaging interval may not include the entire sequence of 13. This points out one advantage of doing the averaging logic in an application, where an adjustment can be made to synchronize the averaging interval to such bursts of beam pulses for display purposes. A periodic request does not specify this kind of “breathing” logic. If it were done, a Vax program might not mind, but the server node might have a problem adjusting to it, since the server node delivers replies to requests at times which depend only on the fixed request period intervals. On the other hand, since the server is not doing the averaging, its last data readings received from the contributing nodes will already be averages. If the server node does not mind the “breathing” in the timing of the contributing nodes, it might be ok. (The server node is not given the job of doing the averaging because it is already a bottleneck by design, and it would require collecting the data from the contrib uting nodes at 15 Hz, while only delivering 1 Hz replies to the consoles. If the averaging is done locally by each contributing node, then the load is distributed; and the load is also much less, because those nodes only send their average values at 1 Hz to the server.) Another value in having the application program support averaging is that the count of the number of beam pulses present in the accumulation of the average can be shown on the display as well. This was done in the olden days for the Linac-only parameter page.

Details

Concentrate on the logic involved in support for the non-server request using the accelerator protocol. (All accelerator protocol logic is in the ACREQS module.) One memory block used in support of these requests is the type#14 internal ptrs block. Each device in a data request is “compiled” into an internal ptr, which is

used to facilitate update of the reply data. Most often, the value of an internal ptr is the memory address of the data to be returned. Specifically, in the case of a request for an analog reading, the internal ptr is the address of the reading data word as it resides in the appropriate field of the ADATA table entry. This makes access to the data value for the purpose of updating the reply data quite trivial.

To support averaging, we still want to maintain a ptr to the reading field, but we must also maintain a longword accumulation value. The allocation of memory needed for the internal ptrs block depends upon the number of internal ptrs needed. We must add an extra longword for each internal ptr that needs to support averaging. During the scan in the NSERVER routine, we should detect the need for this extra space for the internal ptrs block and increase the value of NPTOTAL accordingly, since it is used later to allocate the internal ptrs block. The space for the averaging case will be two longwords per device; the first will be the usual internal ptr value, and the second will be the accumulation longword.

In addition, for the entire request, we need to keep two counters. One is FTDC, the count of cycles over the request period, and the other is SUMC, the number of cycles of data that have been added to produce the accumulation. Also, there must be a state bit that records whether the accumulation holds data from beam cycles or non-beam cycles.

Until now, a call to ACUPDATE from ACUPDCHK produces a new reply to a data request. To support this new feature, we must do some accumulation work *every* cycle, not just the cycles on which a reply is due. A test for the need of accum ulation logic is required in ACUPDCHK before ACUPDATE is called.

During initialization of the non-server request, each device request block (DRB) which needs the special averaging treatment is marked by setting the sign bit of the RDI word in the DRB. (The RDI word contains the read-type routine index and is a small integer used in the READTYPE call to update the answers corres ponding to that DRB.) The test for the need of the averaging logic is that the reading property index (=12 decimal) is used, the listype for analog reading (=0) is used, and the #bytes requested is 2. The internal ptr for this analog reading case will of course be the pointer to the reading word in the analog channel ADATA entry, so the accum ulation logic is simple. If other listypes should need averaging support in the future, they might be additionally permitted. As stated before, the number of internal ptrs required should double for such DRB's to allow room in the internal ptrs block allocated later for the longword needed for the accumulations. Set a flag so that the internal ptrs block header can be marked to indicate that averaging logic is needed by at least one DRB in the request.

For each cycle of a request that needs the averaging logic applied, according to this flag in the internal ptrs block header, the DRB's are scanned in AVGACCUM.

For each DRB marked in its RDI word, accum u lations of the current readings are made according to the averaging algorithm.

Later, during the update scan at the end of the periodic request interval, the sign bit of the RDI word is tested to direct the updating loop to calculate and return the average of the accumulations as the result word, rather than call the usual updating loop. Note that the data values were already accumulated in AVGACCUM before the call to ACUPDATE. All that is necessary is to divide by the number of data values accumulated to obtain the average.

Postscript

This document was a working document used to explore an implementation that would accomplish this averaging feature for the local stations. Most of it was written before the code was started, but it is now updated to reflect what was done to implement the feature. The implementation required 140 lines of source code, about 350 bytes of object code and two days of coding and documen tation effort beyond the initial design discussions and contemplation.

Read-type Routines

Answers from internal ptrs

Oct 12, 1990

A data request made to a local station is “compiled” into an array of internal ptrs. These ptrs are interpreted by “read-type” routines according to the listype(s) used in the request. The read-type routine is called to process an array of internal ptrs that correspond to the array of idents in the original data request and produce the corresponding answers. Because an *array* of internal ptrs is passed to the routine, it is optimized in speed; its logic does the same processing for each element of the array.

As new listypes are added to the system, the support often requires new read-type routines. In the spirit of the named downloaded programs that are used with local applications, this note considers the implementation of named downloaded read-type routines.

Current branch table

The current scheme for selecting the read-type routine involves a table of two-byte offsets to the code (to provide for position independence) for each read-type routine. The read-type# from the listype table is the table index.

As the system grows—the current size is 53K—the use of two-byte offsets may appear limiting, so 4-byte offsets could be used. System changes currently underway will concentrate the knowledge of this branch table into a single routine called READTYPE, whose single argument is the read-type#, so such a change would be simple.

New selection scheme

The new branch table has 4-byte offsets for each resident routine but 4-character names for each downloadable routine. The read-type branch table is scanned. Each offset entry is converted to a ptr to the resident code, and each name entry is converted to a ptr to the executable copy using entries (of type RTYP) found in the CODES table. (For each RTYP entry found, the download copy is sum-checked and copied into a newly-allocated “executable” memory block.) The ptr to the read-type routine is stored in a ram-based table used by READTYPE. There is no provision for replacing or adding a read-type routine without going through the initialization logic at reset time.

Related Groups of Channels

It's all in the family!

Aug 31, 1989

Each analog channel is supported by an analog descriptor entry in the local database. It may be helpful to make associations between channels that are related in some way. A hitherto unused word in the descriptor record has been nominated to fill this need. This note describes its use.

The value stored in the "family" word (at offset 60 bytes in the descriptor, accessible by listype #17) is a delta channel number that indicates the "next" channel in the related group of channels. The value zero then refers to itself. In preparing the family word contents for a group of channels, one should build a chain that links back to itself. In that way, when given any member of the group, one can gain access to all channels in the group. The use of a *delta* channel value means that consecutive channels which all belong to a related group can use the value 1 for the family word. The last channel's family word would likely have a negative value to refer back to the start of the group.

To make accessing the group of related channels easier, a new listype (#49) supports a data request using a channel ident. The returned data consists of a word equal to the number of channels in the group followed by the list of that many channel numbers, beginning with the channel number that was used in the ident. The length of the returned list of channels is also limited to the number of bytes specified in the request. This means that if the size of the group is 9 channels, for example, and the data request is for 16 bytes, the returned data will indicate 7 for the number of channels followed by the first 7 channel numbers in the group.

The motivation for providing this feature was to form a group of channels that all relate to a given timing channel on the Clock Timer board. There are analog channels used for coarse and fine delay control, trigger clock event selection, and clearing all selected trigger events. These can all be formed into a related group, or family, of channels. Given a coarse delay channel, for example, one can ask for the channel group, request the analog control fields for all of the members of the group, and by analysis determine the channels that are used for event selection. The settings of those channels will give the selected clock events for the timing channel.

Restoring a Local Station

Changing its personality

Thu, Dec 8, 1994

Assume that each station's non-volatile memory is periodically saved to a disk file. This could be done on the Vax or the Sun. If one of the operational stations crashes because its non-volatile memory is corrupted, and no other solution is immediately apparent, then there should be a procedure that takes another (non-operational, but working) station and transforms it into the defective one. (One may wish to save the replaceable system memory module first before converting it to replace the memory module in the station that broke.)

First, the station that is broke is not on the network. Take the memory board from a working station and plug it into the broken one in place of its own memory card. When the station is reset upon power-up, it should come up running as it was, but the token ring hardware address is now changed to match the broken station's. With a small change to the system code, we could have the station also come up with the correct IP address, given that we are discussing Linac 133A-based stations. The reason is that all such stations use 131.225.129.xxx for their IP addresses at Fermilab, where xxx is the decimal value of the lo byte of the node# that is sampled from the lo 7 bits of the address switches on the crate utility board, which is a part of all such stations. This address byte would merely be copied into the lo byte of the station's IP address in the header of the IPARP system table.(If one had to change the hi byte of the node#, one might set the hi byte of the hardware address of the "hot spare" via the network, then power it down to get the memory board out to plug into the system to be rebuilt. As soon as the node# is changed, communication would cease, but that's ok because the board would be removed anyway.)

Now, with the station running, we need to cause it to become largely passive and not try to run objectionable local applications. In order to do this from a reset, we need to set a hardware switch accordingly. Also, we don't want the "restore settings" feature to operate after reset, so the restore inhibit switch would also need to be set. Should we consider that the restore inhibit switch is an appropriate means of specifying this "stay low" state? In this state, we would not want the Data Access Table to be active. This will also prevent any local applications from coming alive. Also, we should not want to run a page application, so we should plan to come up on the Index Page in this state. The ADATA and BBYTE tables will remain dormant, and no data will be collected.

Assuming the "stay low" state, what kind of memory restoration can be done?

These tables should already be working, at least nearly:

(put picture #1 here)

These tables can be simply copied directly:

(put picture #2 here)

After all of these tables have been placed, reset the station so it comes up and restores D/A's, with the proper values of settings being sent to the hardware. This will also turn on data streams. Finally, download the RDATA table, which will activate all relevant local applications, start updating the data pool, etc. At this point, the station should be working as the broken one once did when the memory was

saved. Acnet may have save files to download, too, at this point.

Restoring Local Stations

How to return it to life

Mon, Nov 28, 1994

When a local station “dies”, how can one recover it? A procedure must be developed that can be followed by any number of people with only a minimal acquaintance with local station hardware and software. Let us here assume that a station is not working, and it is important to be able to get it back into service as quickly as possible. Possible problems that can exist are too numerous to itemize *fully*, so a basic and reliable starting point must be defined. Assume one starts with a working VME crate, including MVME-133A cpu board, crate utility board, arcnet interface board, token ring network board, and a nonvolatile memory board. Since what is wrong may be the contents of the memory board—not easily verified for total correctness—assume that a generic memory board is available “off-the-shelf.” This note is to be developed into a procedure for restoring a generic memory board to work as the installed one did before.

Generic nonvolatile memory board

Given a board base address of 100000, at location 120000 is normally stored the contents of the operating system code. At location 13E000 is stored the pSOS operating system kernel. This allows for up to 112K of system code, plus 8K of kernel. (At this writing, the size of the system code is about 81K.) The generic board should have this entire space cleared, so that upon reset, a PROM copy of the system code will be entered. (This PROM copy is based at FFF04000.)

Assume that the table directory has already been prepared properly for any Linac local station. The lo byte of the node# is taken from the byte of address switches on the crate utility board. The hi byte is 06 for any Linac local station. (A byte value of 05 is used for diagnostic stations, and 07 is used for D0 stations.) To change the hi byte value via the network means changing the byte at 105046. Once this is done, however, the station should be **reset** to activate the new node#.

Assume that the entry in PAGEP for Page T already points to the PROM-based token ring initialization page application at FFF1A000. This will cause the station to open onto the token ring network after a **reset**.

Assume that an IPARP table already exists for this generic memory board. The only unknown field is the IP address at 10E010. This can be temporarily preset to an IP address that is not normally used. (We assume here that only one such instance of a generic node will be operating at once.) Maybe this could be 131.225.129.9, or 83E1 8109. Note that all token ring nodes at Fermilab use subnet 129, so that their IP addresses are of the form 131.225.129.yyy, or 83E1 81xx. To set the IP address correctly, change the last byte at 10E013 from 09 to xx. After a **reset**, the station should be operating with the correct IP address.

Assume that the generic board already has been loaded with the 09xx table of Acnet IP addresses. This allows the 09EA parameter of the AERS local application to access node OPER via IP to acquire the logical-physical node address table used in Acnet for communication with other Acnet nodes.

How can we determine the correct 09xx for the node that is being initialized? It is stored at 10507E. After changing it, **reset** the station to activate it.

Use the Page G remote access from some host to "log in" to another local station, such as node 0508. Run Page D on that node—in the copy mode—to load up the newly-configured station with local applications and page applications needed by that station. Perhaps we need to keep a record of those needed by each station, along with the parameter values for specific instances of local applications. From a saved memory board file, this information can be extracted.

Many other parameters can already be installed in the generic memory board. These include the token ring group and functional group addresses for reception at 105048, the token ring group addresses for transmission at 105B80, the 'AR' key at 105032 needed to activate use of arcnet, and the data stream table entries for network frame diagnostics and setting log diagnostics.

Reference for saved memory information

In order to provide a proper backup for restoration of the nonvolatile memory board, a copy of it should be saved periodically. From the table directory, one can find out the location of all system tables. The contents of most of these tables can be downloaded to complete the restoration of the system. The procedure for handling restoration of the tables is usually, but not always, simply targeting the generic board with the saved contents of all tables. But some tables need to be treated with special care. See the related document *System Tables and Their Uses* for more details.

System Extensions

Can you do alterations?

Oct 6, 1989

The Local Station system software has evolved over a number of years as new features have been added and changed. This document describes procedures that may be used to make some commonly-requested additions.

Add a new application page

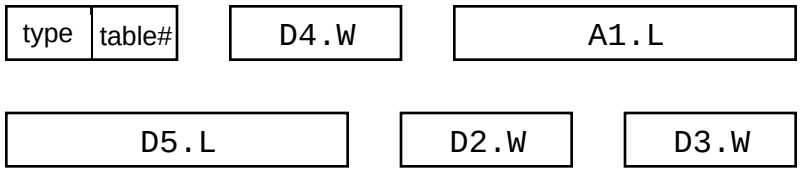
This is done without any system code changes. After preparing the program in S-record format using the cross compiler and/or assembler, find an area in non-volatile memory (so the application will survive a power down) sufficient to contain the linked application. Download the code into that area using the Download Page to process the S-records sent to a serial port of one local station. Go to the index page and find an available page slot to be used in calling up the new application. Invoke the list of entry points of the applications associated with each page by calling up the index page again with the hex switch (one of the small console units buttons) depressed. Enter the starting address—which is assumed to be the entry point—of the application area and press the interrupt button with the cursor just beyond the last character of the 8-digit address *with the hex switch depressed*. (Obviously, without the hex switch depressed, that page would be invoked at the old entry address.) At this point, you should notice that the newly-entered address is displayed as it was typed, and there is no "–" in the second character position of the line. If there is a "–" present, it means that the entry point address does not seem to be valid, and the system will refuse to invoke it. It must be even and not too small, and it must point to a word of memory with the value `$47FA`, the opword for a `LEA disp(A3)` instruction, which must be the first instruction of every application page program. Then call up the new application in the usual way.

To provide a title to the application, call up the page, type the 16-character title on the top line starting in the third character position, and return to the index page by placing the cursor in the home position and interrupting. The new title will be available on the index page and at the top of the application page the next time it is invoked.

Add a new Data Access Table entry type

Design a 16-byte entry format to be used for the new type. The first byte is the type#, a small positive value chosen by checking the branch table READS at the end of the RDADNEW module. The second byte is assumed to be a destination table#, which is usually $00 to denote the ADATA table for analog channel readings or $05 to denote the BBYTE table for binary byte data readings. If there no table# is required, set it to a negative value (like $FF) to denote that it is not a table# to get around the check for the entry# being out of range for the table size. (The auto-setting entry type uses this.) The next two bytes (the second word) are assumed to be the destination table entry#, and is therefore a Chan# (for the ADATA case) or a Byte# (for the BBYTE case).

Write a routine to process the entry and add its entry point to the branch table READS mentioned above. If the routine is external to the RDADNEW module, declare an XREF of course. The routine is called with registers set to the various fields of the 8-word entry as follows:



In addition,

D1.L= offset to the destination table entry in the table
D6.W= #bytes/entry in destination table
A2.L= ptr to destination table entry
Condition codes set by TST.W D3 instruction

All registers may be altered by the routine *except* A3/A5/A6/A7. Examine other routines for examples. The document entitled “RDATA Entry Formats” describes the current entry types available.

Add new Analog Control type

The SETAC module contains many routines selected by the analog control type byte in the analog control field of the analog descriptor. That field is currently 4 bytes in length. The first byte is the type# byte, and the meaning of the other 3 bytes depends upon the type#. A setting to an analog channel device results in a call to one of these routines.

To add a new type of analog control routine, design a data structure that can be used for the analog control field:

Add a new routine reference to both the branch table SETACS and the branch table SETREL at the end of the SETAC module. Write the SETACS routine consistent with the following register-based calling sequence:

D4.W= dataword to be set
A0.L= ptr to analog control field of analog descriptor for this channel
A4.L= ptr to setting word in ADATA table for this channel

Any registers may be altered except A3/A5/A6/A7. By convention, the routine should include copying the dataword into the setting word of the ADATA entry iff no errors are detected in processing the setting. In this way, a readback of the setting value (following the setting command) can determine whether a setting to an analog channel was successful.

To support knob relative settings, the SETREL branch table invokes a routine which scales the knob click, the dataword for the relative setting (listype=7) case, based upon the analog control type. (This may not always be sufficient; the case of 1553 analog control required a separate type# for 12-bit and 16-bit D/A relative control.) The scaled knob click value plus the setting word forms the intended setting value.

Add a new read type routine

An entry in the Listype Table (module LTT) indexed by listype# includes a read type#. The routine indexed by this value is in the READS table at the end of the COLLECT module. (Don't be confused by the name READS also being used in the RDADNEW module; they are different branch tables.) It is invoked by an application program's call to Collect and also by the Update task when updating network requests. (The Server task also calls it using CollectS.)

The routine has a register-based calling sequence. Upon entry to the routine,

D0.W= #idents–1 (or #internal ptrs – 1, since there is one ptr per ident)
D1.W= #bytes to return (>0)
A1.L= Ptr to array of internal ptrs (corresponding to original array of idents)
A2.L= Ptr to data array to be filled

The significance of an internal ptr depends on the code in the REQDGENP or PREQDGEN modules that generated the internal ptr. It is typically a ptr to an entry in a system table, or it may be a ptr into an external answer buffer, usually with the sign bit set to indicate this, or it may be a ptr to a source of zeros—a null ptr. Whatever it is, the read type routine must be aware of its possibilities.

Upon exit from the read type routine, the A2 register must be advanced past the data area of answers produced in satisfying the array of idents. The calling routine then will "even up" the A2 address so that the answers for the next listype, if any, in the request will start on a word boundary. Note that an odd #bytes in a data request will only result in a filler byte after processing the array of idents. Normally, the only odd #bytes likely to be used is 1.

Also upon exit from the read type routine, if the condition code status indicates overflow, the calling routine will assume a bus error occurred during processing and will return an error code 4 to the user.

Besides the A2 register and the condition code status, all registers are available to the read type routine, except A3/A5/A6/A7.

Most of the current read type routines are found in the COLLECT module.

Add a new set type routine

An entry in the Listype Table (module LTT) indexed by listype# includes a set type#. The routine indexed by this value is in the SETS table at the end of the SETDATA module. It is invoked by an application program's call to SetData and also by the Network task when processing setting messages from the network. There are two variations of set type routines. The first is used if the ptr type byte is < 32, indicating a system table#. In this case, the ident is assumed to be a table entry# and is checked to be within the range of the table. The second variation is used if the ptr type byte is ≥ 32.

Upon entry to a set type routine,

D2.W= #bytes of data
D5.W= ptr info byte
D6.W= 0 if short ident, -2 if long ident
A1.L= ptr to data
A2.L= ptr to ident

In addition, if the ptr type < 32, system table parameters are made available as

D4.W= entry# from ident
A4.L= ptr to field in table entry (using D5.W as offset to field in entry)

Add a new ptr type routine

When adding a new read type routine, it is sometimes necessary to add a new ptr type routine as well. The ptr type routine generates an *internal ptr* from an ident. The read type routine generates answer data from an internal ptr. For more explanation of ptr type routines, see the document entitled "Internal Ptrs."

System Tables and Their Uses

Downloadability

Wed, Nov 23, 1994

There are 32 tables defined in a table directory that is located at $100000 in 133A-based stations and $400000 in 162-based stations. This list briefly identifies each in terms of their candidacy for downloading. Tables numbered 0–31 are described.

ADATA EQU 0 ;Analog data

Contains settings and alarm info about all channels. To download settings properly, setting messages should be sent for each controllable channel. If this table is downloaded directly, then a system reset must be performed to cause the settings to be issued to the hardware. Note: This “automatic restore of settings” at reset time can be *inhibited* by the setting of option switch #6, which is on the crate utility board in 133A-based systems, or on the front panel in 162-based IRMs. Under normal conditions, this switch should *not* be set.

ADESC EQU 1 ;Analog descriptors

These records contain scale factors, names, and text for all channels. Can be downloaded record by record. The name hash table is automatically filled during downloading as well as following a system reset. If the table is downloaded directly, this name inserteion is not performed, and a system reset will be necessary to correctly populate the name hash table.

BALRM EQU 2 ;Binary alarm status

These are binary alarm flag words and trip counts for every binary bit. They can be downloaded, if desired. Note: Acnet does not use binary bit-based alarms. But there may be silent bit alarms that are used for keeping diagnostic trip counts. In Acnet, binary alarms are based upon “combined status words.”

BDESC EQU 3 ;Binary bit titles

Every bit has a 16-character title or description in this table. It can be downloaded anytime. Acnet doesn’t use these titles directly, but they serve to describe the meanings of all binary status bits known to the system, so they should be kept up-to-date.

RDATA EQU 4 ;Read Data Access Table

The Data Access Table describes as a list of instructions how to update the local data pool each cycle. It should preferably be downloaded all at once, as it is always “live.”

BBYTE EQU 5 ;Binary raw status bytes

These are the readings of all binary status bytes (8 binary bits per byte) in the system. They are used as setting values to update the digital hardware during

the automatic restore at system reset time. Although one should issue settings for each byte to download this correctly, it is probably easier to perform a system reset.

PAGEP EQU 6 ;Page pointers

The 160-character page titles and associated page program names are kept in this table. It can be downloaded if no page program is active.

PAGEM EQU 7 ;Page private memory

Page context is maintained in this table. It can be downloaded, but it should also bedone only when no page program is active.

LISTP EQU 8 ;Active list pointers

This is a dynamic table of “list#s” associated with actrive data requests. Downloading is neither necessary nor desired.

CODES EQU 9 ;Downloaded named programs

This is a directory for the memory-resident file system that is used to house downloaded page and local applications in non-volatile memory. This cannot be downloaded; rather, it is automatically filled as programs are downloaded via TFTP or copied into the system using the Download page.

CDATA EQU 10 ;Comment alarm data

These records contain alarm flags and counts for comment alarms, such as system reset and alarms reset “events.” Usually these are marked disabled in Acnet.

BADDR EQU 11 ;Binary byte addresses

For each binary byte there is an address that is used for accessing the byte of data and/or controlling it. This table of addresses can be downloaded as needed.

OUTPQ EQU 12 ;Output pointer queue

This is a dynamic table used for a token ring message queue. It should not be downloaded.

PRNTQ EQU 13 ;Print message queue

This is a dynamic table that houses the serial port output queue. It should not be downloaded.

LATBL EQU 14 ;Local applications

This table holds the list of local application instances and associated parameter values. To download this table, first insure that all local applications presently running, if any, are disabled. Each entry to be downloaded must be

modified to have a null static variables pointer and the “last status” cleared to indicate inactive.

CPROQ EQU 15 ;Co-processor message queue

This table's records identify the location and size of coprocessor message queues located in shared memory. They are not used in Acnet stations.

MMAPS EQU 16 ;Memory-mapped templates

Entries in this table support selective downloading into special hardware memory boards via vertical interconnect. It is used only for D0, which probably already has a means of downloading the needed board templates.

Q1553 EQU 17 ;1553 controller queue ptrs

Dynamic table for D0 1553 rack monitors. Downloading inappropriate.

DSTRM EQU 18 ;Data streams

Defines location and sizes of data streams. Entry #0 and #1 are standard and used in all stations. #0 provides for network frame diagnostics. #1 supports a settings log. Additional data streams may be defined and downloaded for other purposes. Extensively used in D0 high voltage systems. A system reset is required to initialize the related data stream queues.

SERIQ EQU 19 ;Serial Input Queue

Dynamic queue of serial input data. Downloading inappropriate.

TBL20 EQU 20 ;spare (used by MSU?)

Undefined.

AADIB EQU 21 ;Analog alarm device info block (D0)

Analog alarm device info used in D0 downloadable by D0 utility.

BADIB EQU 22 ;Binary alarm device info block (D0)

Binary alarm device info used in D0 downloadable by D0 utility.

CADIB EQU 23 ;Comment alarm device info block (D0)

Comment alarm device info used in D0 downloadable by D0 utility.

DSTAT EQU 24 ;Combined status words specifications (Acnet)

Table entries define construction of 16-bit word binary status words. Table can be downloaded. Entries are referenced by Data Access Table entries type $26.

TBL25 EQU 25 ;spare

Undefined.

TBL26 EQU 26 ;spare
Undefined.

IPNAT EQU 27 ;IP Node Address Table (node#s, IP addresses)
This table is a cache of IP addresses received from DNS queries based on nade# names of the form NODE0576, for example. It can be downloaded. The important part is the header that includes the IP address of the DNS node and an up-to-16-character default suffix for node names NODExxxx.

IPARP EQU 28 ;IP “ARP” Table, incl IP security table
From scratch, this Internet Protocol table’s header is automatically initialized at reset time. Then additional info must be filled in, such as station’s IP address, subnet mask, gateway IP address and MTU’s. A system reset must then be performed to activate IP support.

DIAGQ EQU 29 ;Alloc,Liber diagnostic queue (optional)
Downloading inappropriate.

TRING EQU 30 ;Token Ring Network Table
Some parts of this table can be downloaded. It provides much of the basic network, especially token ring, support.

DLOAD EQU 31 ;Download area (last word: table directory entry cksm).
From scratch, initialized empty at system reset time. Automatically filled by programs that are downloaded via TFTP or copied via the Download page. Downloadingthis “table” directly is inappropriate. Usually 192K bytes in size.